PYTHON 3

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985 – 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). Python is named after a TV Show called 'Monty Python's Flying Circus' and not after Python-the snake.

Python 3.0 was released in 2008. Although this version is supposed to be backward incompatibles, later on many of its important features have been backported to be compatible with the version 2.7. This tutorial gives enough understanding on Python 3 version programming language. Please refer to this link for our Python 2 tutorial.

Audience

This tutorial is designed for software programmers who want to upgrade their Python skills to Python 3. This tutorial can also be used to learn Python programming language from scratch.

Prerequisites

You should have a basic understanding of Computer Programming terminologies. A basic understanding of any of the programming languages is a plus.

Execute Python Programs

For most of the examples given in this tutorial you will find **Try it** option, so just make use of it and enjoy your learning.

Try the following example using **Try it** option available at the top right corner of the below sample code box –

```
#!/usr/bin/python3

print ("Hello, Python!")
```

Copyright & Disclaimer

Table of Contents

Python 3 – Basic Tutorial

1. Python 3 – What is New?

The __future__ module

Python 3.x introduced some Python 2-incompatible keywords and features that can be imported via the in-built __future__ module in Python 2. It is recommended to use __future__ imports, if you are planning Python 3.x support for your code.

For example, if we want Python 3.x's integer division behavior in Python 2, add the following import statement.

```
from __future__ import division
```

The print Function

Most notable and most widely known change in Python 3 is how the **print** function is used. Use of parenthesis () with print function is now mandatory. It was optional in Python 2.

```
print "Hello World" #is acceptable in Python 2

print ("Hello World") # in Python 3, print must be followed by ()
```

The print() function inserts a new line at the end, by default. In Python 2, it can be suppressed by putting ',' at the end. In Python 3, "end=' '" appends space instead of newline.

```
print x,           # Trailing comma suppresses newline in Python 2

print(x, end=" ")  # Appends a space instead of a newline in Python 3
```

Reading Input from Keyboard

Python 2 has two versions of input functions, **input()** and **raw_input()**. The input() function treats the received data as string if it is included in quotes '' or "", otherwise the data is treated as number.

In Python 3, raw_input() function is deprecated. Further, the received data is always treated as string.

```
In Python 2

>>> x=input('something:')

something:10 #entered data is treated as number

>>> x

10

>>> x=input('something:')

something:'10' #eentered data is treated as string
```

```
>>> x
'10'
>>> x=raw_input("something:")
something:10 #entered data is treated as string even without ''
>>> x
'10'
>>> x=raw_input("something:")
something:'10' #entered data treated as string including ''
>>> x
"'10'"
In Python 3
>>> x=input("something:")
something:10
>>> x
'10'
>>> x=input("something:")
something:'10' #entered data treated as string with or without ''
>>> x
"'10'"
>>> x=raw_input("something:") # will result NameError
Traceback (most recent call last):
  File "", line 1, in

    x=raw_input("something:")
NameError: name 'raw_input' is not defined
```

Integer Division

In Python 2, the result of division of two integers is rounded to the nearest integer. As a result, 3/2 will show 1. In order to obtain a floating-point division, numerator or denominator must be explicitly used as float. Hence, either 3.0/2 or 3/2.0 or 3.0/2.0 will result in 1.5

Python 3 evaluates 3 / 2 as 1.5 by default, which is more intuitive for new programmers.

Unicode Representation

Python 2 requires you to mark a string with a **u** if you want to store it as Unicode.

Python 3 stores strings as Unicode, by default. We have Unicode (utf-8) strings, and 2 byte classes: byte and byte arrays.

xrange() Function Removed

In Python 2 range() returns a list, and xrange() returns an object that will only generate the items in the range when needed, saving memory.

In Python 3, the range() function is removed, and xrange() has been renamed as range(). In addition, the range() object supports slicing in Python 3.2 and later .

raise excepríon

Python 2 accepts both notations, the 'old' and the 'new' syntax; Python 3 raises a SyntaxError if we do not enclose the exception argument in parenthesis.

```
raise IOError, "file error" #This is accepted in Python 2
raise IOError("file error") #This is also accepted in Python 2
raise IOError, "file error" #syntax error is raised in Python 3
raise IOError("file error") #this is the recommended syntax in Python 3
```

Arguments in Exceptions

In Python 3, arguments to exception should be declared with 'as' keyword.

```
except Myerror, err: # In Python2
except Myerror as err: #In Python 3
```

next() Function and .next() Method

In Python 2, next() as a method of generator object, is allowed. In Python 2, the next() function, to iterate over generator object, is also accepted. In Python 3, however, next(0 as a generator method is discontinued and raises **AttributeError.**

```
gen = (letter for letter in 'Hello World') # creates generator object
next(my_generator) #allowed in Python 2 and Python 3
my_generator.next() #allowed in Python 2. raises AttributeError in Python 3
```

2to3 Utility

Along with Python 3 interpreter, 2to3.py script is usually installed in tools/scripts folder. It reads Python 2.x source code and applies a series of fixers to transform it into a valid Python 3.x code.

```
Here is a sample Python 2 code (area.py):
def area(x,y=3.14):
    a=y*x*x
    print a
    return a
```

```
a=area(10)
print "area",a
To convert into Python 3 version:
$2to3 -w area.py
Converted code :
def area(x,y=3.14): # formal parameters
    a=y*x*x
    print (a)
    return a
a=area(10)
print("area",a)
```

2. Python 3 – Overview

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas the other languages use punctuations. It has fewer syntactical constructions than other languages.

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.

- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Python Features

Python's features include-

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.

- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.

- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.

- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

- **Interactive Mode:** Python has support for an interactive mode, which allows interactive testing and debugging of snippets of code.

- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

- **Databases:** Python provides interfaces to all major commercial databases.

- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features. A few are listed below-

- It supports functional and structured programming methods as well as OOP.

- It can be used as a scripting language or can be compiled to byte-code for building large applications.

- It provides very high-level dynamic data types and supports dynamic type checking.

- It supports automatic garbage collection.

- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

3. Python 3 – Environment Setup

Try it Option Online

We have set up the Python Programming environment online, so that you can compile and execute all the available examples online. It will give you the confidence in what you are reading and will enable you to verify the programs with different options. Feel free to modify any example and execute it online.

Try the following example using our online compiler available at CodingGround

```
#!/usr/bin/python3

print ("Hello, Python!")
```

For most of the examples given in this tutorial, you will find a **Try it** option on our website code sections, at the top right corner that will take you to the online compiler. Just use it and enjoy your learning.

Python 3 is available for Windows, Mac OS and most of the flavors of Linux operating system. Even though Python 2 is available for many other OSs, Python 3 support either has not been made available for them or has been dropped.

Local Environment Setup

Open a terminal window and type "python" to find out if it is already installed and which version is installed.

Getting Python

Windows platform

Binaries of latest version of Python 3 (Python 3.5.1) are available on this download page

The following different installation options are available.

- Windows x86-64 embeddable zip file
- Windows x86-64 executable installer
- Windows x86-64 web-based installer
- Windows x86 embeddable zip file
- Windows x86 executable installer
- Windows x86 web-based installer

Note: In order to install Python 3.5.1, minimum OS requirements are Windows 7 with SP1. For versions 3.0 to 3.4.x, Windows XP is acceptable.

Linux platform

Different flavors of Linux use different package managers for installation of new packages.

On Ubuntu Linux, Python 3 is installed using the following command from the terminal.

```
$sudo apt-get install python3-minimal
```

Installation from source

```
Download Gzipped source tarball from Python's download URL:
https://www.python.org/ftp/python/3.5.1/Python-3.5.1.tgz

Extract the tarball

tar xvfz Python-3.5.1.tgz

Configure and Install:

cd Python-3.5.1

./configure --prefix=/opt/python3.5.1

make

sudo make install
```

Mac OS

Download Mac OS installers from this URL:https://www.python.org/downloads/mac-osx/

- Mac OS X 64-bit/32-bit installer : python-3.5.1-macosx10.6.pkg
- Mac OS X 32-bit i386/PPC installer : python-3.5.1-macosx10.5.pkg

Double click this package file and follow the wizard instructions to install.

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python:

Python Official Website : http://www.python.org/

You can download Python documentation from the following site. The documentation is available in HTML, PDF and PostScript formats.

Python Documentation Website : www.python.org/doc/

Setting up PATH

Programs and other executable files can be in many directories. Hence, the operating systems provide a search path that lists the directories that it searches for executables.

The important features are-

- The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

- The path variable is named as **PATH** in Unix or **Path** in Windows (Unix is case-sensitive; Windows is not).
- In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Setting Path at Unix/Linux

To add the Python directory to the path for a particular session in Unix-

- **In the csh shell:** type setenv PATH "$PATH:/usr/local/bin/python3" and press Enter.
- **In the bash shell (Linux):** type export PATH="$PATH:/usr/local/bin/python3" and press Enter.
- **In the sh or ksh shell:** type PATH="$PATH:/usr/local/bin/python3" and press Enter.

Note: /usr/local/bin/python3 is the path of the Python directory.

Setting Path at Windows

To add the Python directory to the path for a particular session in Windows-

At the command prompt : type
path %path%;C:\Python and press Enter.

Note: C:\Python is the path of the Python directory.

Python Environment Variables

Here are important environment variables, which are recognized by Python-

Variable	Description
PYTHONPATH	It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes, preset by the Python installer.
PYTHONSTARTUP	It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.

PYTHONCASEOK	It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
PYTHONHOME	It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Running Python

There are three different ways to start Python-

(1) Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter **python** the command line.

Start coding right away in the interactive interpreter.

```
$python             # Unix/Linux
or
python%             # Unix/Linux
or
C:>python           # Windows/DOS
```

Here is the list of all the available command line options-

Option	Description
-d	provide debug output
-O	generate optimized bytecode (resulting in .pyo files)
-S	do not run import site to look for Python paths on startup
-v	verbose output (detailed trace on import statements)
-X	disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6
-c cmd	run Python script sent in as cmd string

file	run Python script from given file

(2) Script from the Command-line

A Python script can be executed at the command line by invoking the interpreter on your application, as shown in the following example.

```
$python  script.py          # Unix/Linux

or

python% script.py           # Unix/Linux

or

C:>python script.py         # Windows/DOS
```

Note: Be sure the file permission mode allows execution.

(3) Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix:** IDLE is the very first Unix IDE for Python.
- **Windows: PythonWin** is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

If you are not able to set up the environment properly, then you can take the help of your system admin. Make sure the Python environment is properly set up and working perfectly fine.

Note: All the examples given in subsequent chapters are executed with Python 3.4.1 version available on Windows 7 and Ubuntu Linux.

We have already set up Python Programming environment online, so that you can execute all the available examples online while you are learning theory. Feel free to modify any example and execute it online.

4. Python 3 – Basic Syntax

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

First Python Program

Let us execute the programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt-

```
$ python
Python 3.3.2 (default, Dec 10 2013, 11:35:01)
[GCC 4.6.3] on Linux
Type "help", "copyright", "credits", or "license" for more information.
>>>
On Windows:
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Type the following text at the Python prompt and press Enter-

```
>>> print ("Hello, Python!")
```

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. This produces the following result-

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have the extension**.py**. Type the following source code in a test.py file-

```
print ("Hello, Python!")
```

We assume that you have the Python interpreter set in **PATH** variable. Now, try to run this program as follows-

On Linux

```
$ python test.py
```

This produces the following result-

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result-

```
Hello, Python!
```

Let us try another way to execute a Python script in Linux. Here is the modified test.py file-

```
#!/usr/bin/python3
print ("Hello, Python!")
```

We assume that you have Python interpreter available in the /usr/bin directory. Now, try to run this program as follows-

```
$ chmod +x test.py     # This is to make file executable
$./test.py
```

This produces the following result-

```
Hello, Python!
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers-

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.

- Starting an identifier with two leading underscores indicates a strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	Not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Lines and Indentation

Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example-

```
if True:
    print ("True")
else:
  print ("False")
```

However, the following block generates an error-

```
if True:
    print ("Answer")
    print ("True")
else:
    print "(Answer")
  print ("False")
```

Thus, in Python all the continuous lines indented with the same number of spaces would form a block. The following example has various statement blocks-

Note: Do not try to understand the logic at this point of time. Just make sure you understood the various blocks even if they are without braces.

```
#!/usr/bin/python3
import sys
try:
  # open file stream
  file = open(file_name, "w")
except IOError:
  print ("There was an error writing to", file_name)
  sys.exit()
print ("Enter '", file_finish,)
print "' When finished"
while file_text != file_finish:
  file_text = raw_input("Enter text: ")
  if file_text == file_finish:
    # close the file
    file.close
    break
  file.write(file_text)
  file.write("\n")
file.close()
file_name = input("Enter filename: ")
if len(file_name) == 0:
  print ("Next time please enter something")
```

```
  sys.exit()
try:
  file = open(file_name, "r")
except IOError:
  print ("There was an error reading file")
  sys.exit()
file_text = file.read()
file.close()
print (file_text)
```

Multi-Line Statements

Statements in Python typically end with a new line. Python, however, allows the use of the line continuation character (\) to denote that the line should continue. For example-

```
total = item_one + \
        item_two + \
        item_three
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example-

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal-

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python3
```

```
# First comment
print ("Hello, Python!") # second comment
```

This produces the following result-

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression-

```
name = "Madisetti" # This is again comment
```

Python does not have multiple-line commenting feature. You have to comment each line individually as follows-

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt and the statement saying "Press the enter key to exit", and then waits for the user to take action –

```
#!/usr/bin/python3
input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block. Here is a sample snip using the semicolon-

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

Groups of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example –

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with **-h**:

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit
[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advance topic. Let us understand it.

Command Line Arguments

Python provides a **getopt** module that helps you parse command-line options and arguments.

```
$ python test.py arg1 arg2 arg3
```

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purposes-

- **sys.argv** is the list of command-line arguments.
- **len(sys.argv)** is the number of command-line arguments.

Here sys.argv[0] is the program i.e. the script name.

Example

Consider the following script **test.py-**

```
#!/usr/bin/python3
import sys
print ('Number of arguments:', len(sys.argv), 'arguments.')
print ('Argument List:', str(sys.argv))
```

Now run the above script as follows –

```
$ python test.py arg1 arg2 arg3
```

This produces the following result-

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

NOTE: As mentioned above, the first argument is always the script name and it is also being counted in number of arguments.

Parsing Command-Line Arguments

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command line argument parsing.

getopt.getopt method

This method parses the command line options and parameter list. Following is a simple syntax for this method-

```
getopt.getopt(args, options, [long_options])
```

Here is the detail of the parameters-

- **args**: This is the argument list to be parsed.
- **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon (:).
- **long_options**: This is an optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.
- This method returns a value consisting of two elements- the first is a list of **(option, value)** pairs, the second is a list of program arguments left after the option list was stripped.

- Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., '-x') or two hyphens for long options (e.g., '--long-option').

Exception getopt.GetoptError

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option.

Example

Suppose we want to pass two file names through command line and we also want to give an option to check the usage of the script. Usage of the script is as follows-

```
usage: test.py -i <inputfile> -o <outputfile>
```

Here is the following script to test.py-

```
#!/usr/bin/python3
import sys, getopt
def main(argv):
   inputfile = ''
   outputfile = ''
   try:
      opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
   except getopt.GetoptError:
      print ('test.py -i <inputfile> -o <outputfile>')
      sys.exit(2)
   for opt, arg in opts:
      if opt == '-h':
         print ('test.py -i <inputfile> -o <outputfile>')
         sys.exit()
      elif opt in ("-i", "--ifile"):
         inputfile = arg
      elif opt in ("-o", "--ofile"):
         outputfile = arg
   print ('Input file is "', inputfile)
   print ('Output file is "', outputfile)
if __name__ == "__main__":
   main(sys.argv[1:])
```

Now, run the above script as follows-

```
$ test.py -h
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i BMP -o
usage: test.py -i <inputfile> -o <outputfile>
$ test.py -i inputfile -o outputfile
Input file is " inputfile
Output file is " outputfile
```

5. Python 3 – Variable Types

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to the variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example-

```
#!/usr/bin/python3
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result –

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously.

For example-

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example-

```
a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to the variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types-

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example-

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the **del** statement. The syntax of the **del** statement is –

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the **del** statement.

For example-

```
del var
del var_a, var_b
```

Python supports three different numerical types –

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

All integers in Python 3 are represented as long integers. Hence, there is no separate number type as long.

Examples

Here are some examples of numbers-

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example-

```
#!/usr/bin/python3
str = 'Hello World!'
print (str)          # Prints complete string
print (str[0])       # Prints first character of the string
print (str[2:5])     # Prints characters starting from 3rd to 5th
print (str[2:])      # Prints string starting from 3rd character
print (str * 2)      # Prints string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result-

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example-

```
#!/usr/bin/python3
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list)          # Prints complete list
print (list[0])       # Prints first element of the list
print (list[1:3])     # Prints elements starting from 2nd till 3rd
print (list[2:])      # Prints elements starting from 3rd element
print (tinylist * 2)  # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

This produces the following result-

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis.

The main difference between lists and tuples is- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example-

```
#!/usr/bin/python3
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print (tuple)           # Prints complete tuple
print (tuple[0])        # Prints first element of the tuple
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])       # Prints elements starting from 3rd element
print (tinytuple * 2)   # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

This produces the following result-

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
#!/usr/bin/python3
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash-table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example-

```
#!/usr/bin/python3
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one'])       # Prints value for 'one' key
print (dict[2])           # Prints value for 2 key
print (tinydict)          # Prints complete dictionary
print (tinydict.keys())   # Prints all the keys
print (tinydict.values()) # Prints all the values
```

This produces the following result-

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among the elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type-name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Function	Description
int(x [,base])	Converts x to an integer. The base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.

str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

6. Python 3 – Basic Operators

Operators are the constructs, which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called the operator.

Types of Operator

Python language supports the following types of operators-

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look at all the operators one by one.

Python Arithmetic Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then-

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20

//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Example

Assume variable a holds 10 and variable b holds 20, then-

```
#!/usr/bin/python3
a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ", c)

c = a - b
print ("Line 2 - Value of c is ", c )

c = a * b
print ("Line 3 - Value of c is ", c)

c = a / b
print ("Line 4 - Value of c is ", c )

c = a % b
print ("Line 5 - Value of c is ", c)

a = 2
b = 3
c = a**b
print ("Line 6 - Value of c is ", c)

a = 10
b = 5
c = a//b
print ("Line 7 - Value of c is ", c)
```

When you execute the above program, it produces the following result-

```
Line 1 - Value of c is  31
Line 2 - Value of c is  11
```

```
Line 3 - Value of c is  210
Line 4 - Value of c is  2.1
Line 5 - Value of c is  1
Line 6 - Value of c is  8
Line 7 - Value of c is  2
```

Python Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds the value 10 and variable b holds the value 20, then-

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example

Assume variable a holds 10 and variable b holds 20, then-

```
#!/usr/bin/python3
a = 21
b = 10
if ( a == b ):
   print ("Line 1 - a is equal to b")
else:
```

```
   print ("Line 1 - a is not equal to b")

if ( a != b ):
   print ("Line 2 - a is not equal to b")
else:
   print ("Line 2 - a is equal to b")

if ( a < b ):
   print ("Line 3 - a is less than b" )
else:
   print ("Line 3 - a is not less than b")

if ( a > b ):
   print ("Line 4 - a is greater than b")
else:
   print ("Line 4 - a is not greater than b")

a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

if ( a <= b ):
   print ("Line 5 - a is either less than or equal to  b")
else:
   print ("Line 5 - a is neither less than nor equal to  b")

if ( b >= a ):
   print ("Line 6 - b is either greater than  or equal to b")
else:
   print ("Line 6 - b is neither greater than  nor equal to b")
```

When you execute the above program, it produces the following result-

```
Line 1 - a is not equal to b
Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to  b
Line 6 - b is either greater than  or equal to b
```

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then-

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a

-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example

Assume variable a holds 10 and variable b holds 20, then-

```
#!/usr/bin/python3

a = 21
b = 10
c = 0

c = a + b
print ("Line 1 - Value of c is ", c)

c += a
print ("Line 2 - Value of c is ", c )

c *= a
print ("Line 3 - Value of c is ", c )
```

```
c /= a
print ("Line 4 - Value of c is ", c )

c  = 2
c %= a
print ("Line 5 - Value of c is ", c)

c **= a
print ("Line 6 - Value of c is ", c)

c //= a
print ("Line 7 - Value of c is ", c)
```

When you execute the above program, it produces the following result-

```
Line 1 - Value of c is  31
Line 2 - Value of c is  52
Line 3 - Value of c is  1092
Line 4 - Value of c is  52.0
Line 5 - Value of c is  2
Line 6 - Value of c is  2097152
Line 7 - Value of c is  99864
```

Python Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows-

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Pyhton's built-in function bin() can be used to obtain binary representation of an integer number.

The following Bitwise operators are supported by Python language-

Operator	Description	Example
& Binary AND	Operator copies a bit to the result, if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Example

```
#!/usr/bin/python3

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
print ('a=',a,':',bin(a),'b=',b,':',bin(b))
c = 0

c = a & b;        # 12 = 0000 1100
print ("result of AND is ", c,':',bin(c))

c = a | b;        # 61 = 0011 1101
print ("result of OR is ", c,':',bin(c))
```

```
c = a ^ b;        # 49 = 0011 0001
print ("result of EXOR is ", c,':',bin(c))

c = ~a;           # -61 = 1100 0011
print ("result of COMPLEMENT is ", c,':',bin(c))

c = a << 2;       # 240 = 1111 0000
print ("result of LEFT SHIFT is ", c,':',bin(c))

c = a >> 2;       # 15 = 0000 1111
print ("result of RIGHT SHIFT is ", c,':',bin(c))
```

When you execute the above program, it produces the following result-

```
a= 60 : 0b111100 b= 13 : 0b1101
result of AND is  12 : 0b1100
result of OR is  61 : 0b111101
result of EXOR is  49 : 0b110001
result of COMPLEMENT is  -61 : -0b111101
result of LEFT SHIFT is  240 : 0b11110000
result of RIGHT SHIFT is  15 : 0b111
```

Python Logical Operators

The following logical operators are supported by Python language. Assume variable a holds True and variable b holds False then-

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below-

Operator	Description	Example
in	Evaluates to true, if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true, if it does not find a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Example

```
#!/usr/bin/python3

a = 10
b = 20
list = [1, 2, 3, 4, 5 ]

if ( a in list ):
   print ("Line 1 - a is available in the given list")
else:
   print ("Line 1 - a is not available in the given list")

if ( b not in list ):
   print ("Line 2 - b is not available in the given list")
else:
   print ("Line 2 - b is available in the given list")

c=b/a
if ( c in list ):
   print ("Line 3 - a is available in the given list")
else:
print ("Line 3 - a is not available in the given list")
```

When you execute the above program, it produces the following result-

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Example

```
#!/usr/bin/python3

a = 20
b = 20
print ('Line 1','a=',a,':',id(a), 'b=',b,':',id(b))

if ( a is b ):
   print ("Line 2 - a and b have same identity")
else:
   print ("Line 2 - a and b do not have same identity")

if ( id(a) == id(b) ):
   print ("Line 3 - a and b have same identity")
else:
   print ("Line 3 - a and b do not have same identity")
```

```
b = 30
print ('Line 4','a=',a,':',id(a), 'b=',b,':',id(b))

if ( a is not b ):
   print ("Line 5 - a and b do not have same identity")
else:
   print ("Line 5 - a and b have same identity")
```

When you execute the above program, it produces the following result-

```
Line 1 a= 20 : 1594701888 b= 20 : 1594701888
Line 2 - a and b have same identity
Line 3 - a and b have same identity
Line 4 a= 20 : 1594701888 b= 30 : 1594702048
Line 5 - a and b do not have same identity
```

Python Operators Precedence

The following table lists all the operators from highest precedence to the lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators

= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator precedence affects the evaluation of an an expression.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because the operator * has higher precedence than +, so it first multiplies 3*2 and then is added to 7.

Here, the operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Example

```
#!/usr/bin/python3

a = 20
b = 10
c = 15
d = 5

print ("a:%d b:%d c:%d d:%d" % (a,b,c,d ))
e = (a + b) * c / d       #( 30 * 15 ) / 5
print ("Value of (a + b) * c / d is ",  e)

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print ("Value of ((a + b) * c) / d is ",  e)

e = (a + b) * (c / d)    # (30) * (15/5)
print ("Value of (a + b) * (c / d) is ",  e)

e = a + (b * c) / d      #  20 + (150/5)
print ("Value of a + (b * c) / d is ",  e)
```

When you execute the above program, it produces the following result-

```
a:20 b:10 c:15 d:5
Value of (a + b) * c / d is  90.0
```

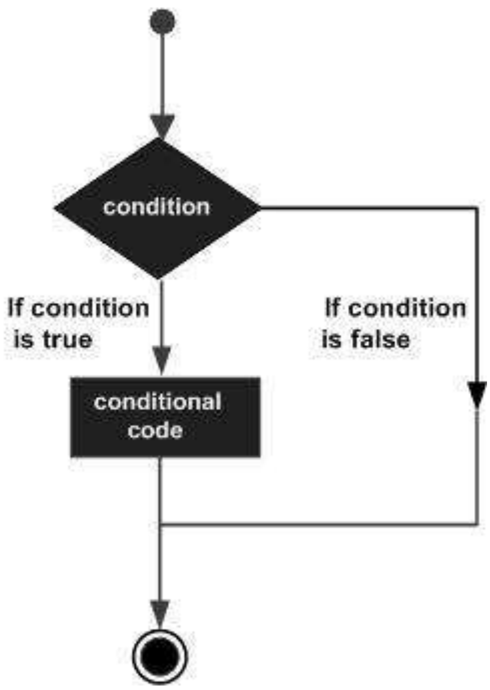
```
Value of ((a + b) * c) / d is  90.0
Value of (a + b) * (c / d) is  90.0
Value of a + (b * c) / d is  50.0
```

7. Python 3 – Decision Making

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.

Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages-



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and any **zero** or **null values** as FALSE value.

Python programming language provides the following types of decision-making statements.

Statement	Description
if statements	An if statement consists of a Boolean expression followed by one or more statements.
if...else statements	An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

nested if statements	You can use one if or else if statement inside another if or else if statement(s).

Let us go through each decision-making statement quickly.

IF Statement

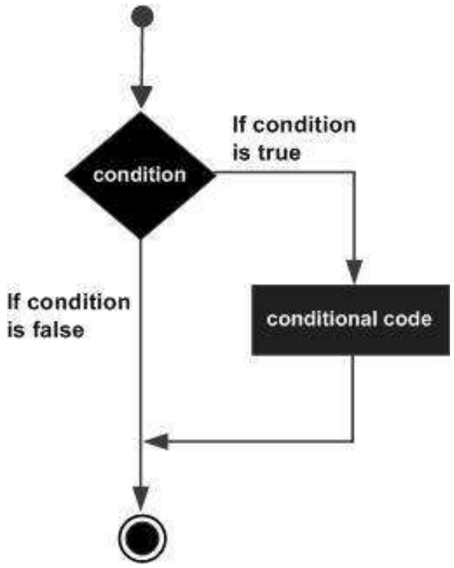
The IF statement is similar to that of other languages. The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

Syntax

```
if expression:
   statement(s)
```

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Flow Diagram



Example

```
#!/usr/bin/python3
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)
```

```
var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

When the above code is executed, it produces the following result –

```
1 - Got a true expression value
100
Good bye!
```

IF...ELIF...ELSE Statements

An **else** statement can be combined with an **if** statement. An **else** statement contains a block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

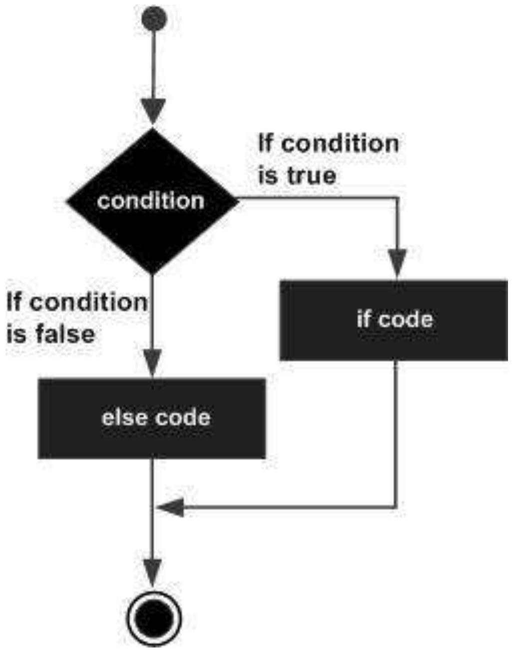
The else statement is an optional statement and there could be at the most only one **else** statement following **if**.

Syntax

The syntax of the **if...else** statement is-

```
if expression:
   statement(s)
else:
   statement(s)
```

Flow Diagram



Example

```
#!/usr/bin/python3
amount=int(input("Enter amount: "))
if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
else:
    discount=amount*0.10
    print ("Discount",discount)

print ("Net payable:",amount-discount)
```

In the above example, discount is calculated on the input amount. Rate of discount is 5%, if the amount is less than 1000, and 10% if it is above 10000. When the above code is executed, it produces the following result-

```
Enter amount: 600
Discount 30.0
Net payable: 570.0
Enter amount: 1200
Discount 120.0
```

```
Net payable: 1080.0
```

The elif Statement

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at the most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows-

Example

```
#!/usr/bin/python3
amount=int(input("Enter amount: "))

if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount=amount*0.10
    print ("Discount",discount)
else:
    discount=amount*0.15
    print ("Discount",discount)
print ("Net payable:",amount-discount)
```

When the above code is executed, it produces the following result-

```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 3000
Discount 300.0
Net payable: 2700.0

Enter amount: 6000
Discount 900.0
Net payable: 5100.0
```

Nested IF Statements

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

The syntax of the nested if...elif...else construct may be-

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example

```
# !/usr/bin/python3
num=int(input("enter number"))
```

```
if num%2==0:
    if num%3==0:
        print ("Divisible by 3 and 2")
    else:
        print ("divisible by 2 not divisible by 3")
else:
    if num%3==0:
        print ("divisible by 3 not divisible by 2")
    else:
        print  ("not Divisible by 2 not divisible by 3")
```

When the above code is executed, it produces the following result-

```
enter number8
divisible by 2 not divisible by 3

enter number15
divisible by 3 not divisible by 2

enter number12
Divisible by 3 and 2

enter number5
not Divisible by 2 not divisible by 3
```

Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause-

```
#!/usr/bin/python3
var = 100
if ( var  == 100 ) : print ("Value of expression is 100")
print ("Good bye!")
```

When the above code is executed, it produces the following result-

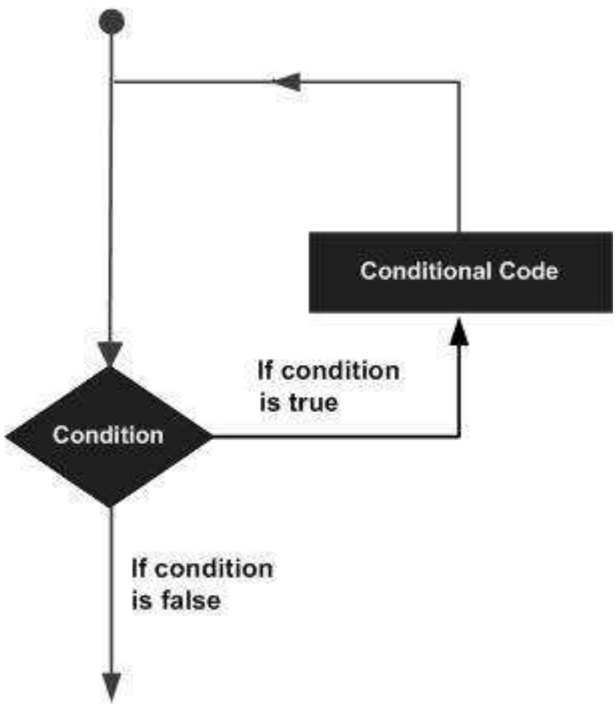
```
Value of expression is 100
Good bye!
```

8. Python 3 – Loops

In general, statements are executed sequentially- The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement.



Python programming language provides the following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops	You can use one or more loop inside any another while, or for loop.

while Loop Statements

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is-

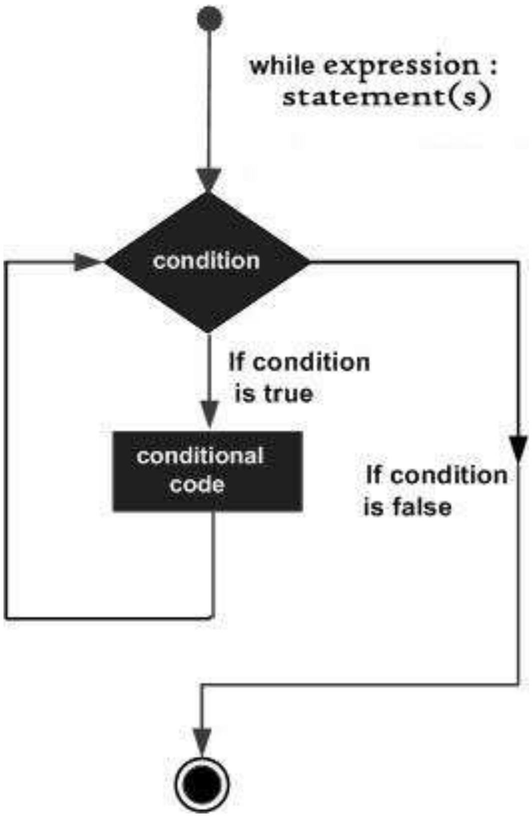
```
while expression:
   statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements with uniform indent. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, a key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#!/usr/bin/python3


count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1


print ("Good bye!")
```

When the above code is executed, it produces the following result-

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
```

```
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
#!/usr/bin/python3
var = 1
while var == 1 :  # This constructs an infinite loop
   num = int(input("Enter a number  :"))
   print ("You entered: ", num)
print ("Good bye!")
```

When the above code is executed, it produces the following result-

```
Enter a number  :20
You entered:  20
Enter a number  :29
You entered:  29
Enter a number  :3
You entered:  3
Enter a number  :11
You entered:  11
Enter a number  :22
You entered:  22
Enter a number  :Traceback (most recent call last):
  File "examples\test.py", line 5, in
    num = int(input("Enter a number  :"))
KeyboardInterrupt
```

The above example goes in an infinite loop and you need to use CTRL+C to exit the program.

Using else Statement with Loops

Python supports having an **else** statement associated with a loop statement.

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise the else statement gets executed.

```
#!/usr/bin/python3
count = 0
while count < 5:
   print (count, " is  less than 5")
   count = count + 1
else:
   print (count, " is not less than 5")
```

When the above code is executed, it produces the following result-

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

Single Statement Suites

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a **one-line while** clause-

```
#!/usr/bin/python3
flag = 1
while (flag): print ('Given flag is really true!')
print ("Good bye!")
```

The above example goes into an infinite loop and you need to press CTRL+C keys to exit.

for Loop Statements

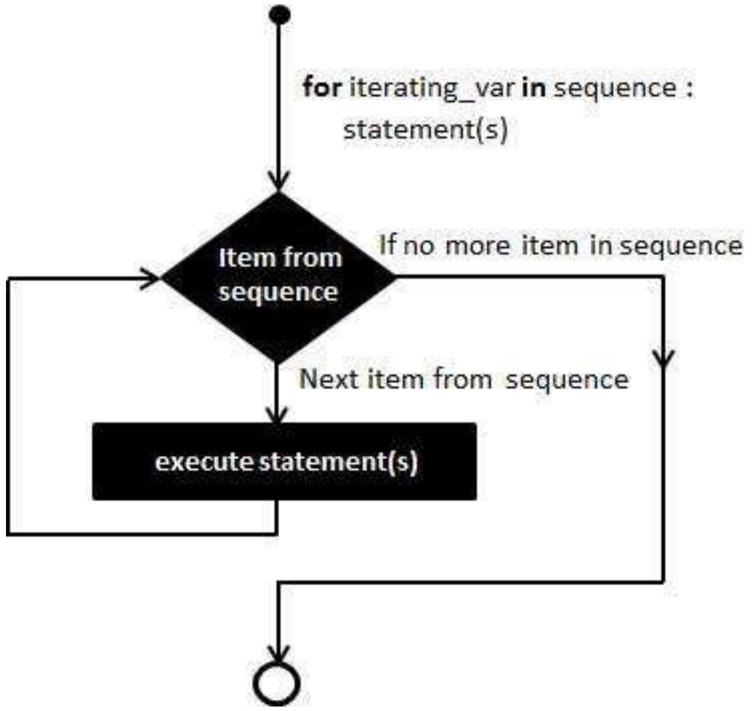
The for statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax

```
for iterating_var in sequence:
   statements(s)
```

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



The range() function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

range() generates an iterator to progress integers starting with 0 upto n-1. To obtain a list object of the sequence, it is typecasted to list(). Now this list can be iterated using the for statement.

```
>>> for var in list(range(5)):
     print (var)
```

This will produce the following output.

```
0
1
2
3
4
```

Example

```
#!/usr/bin/python3
for letter in 'Python':     # traversal of a string sequence
   print ('Current Letter :', letter)
print()
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
   print ('Current fruit :', fruit)

print ("Good bye!")
```

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
```

```
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n

Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example-

```
#!/usr/bin/python3
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])
print ("Good bye!")
```

When the above code is executed, it produces the following result-

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with Loops

Python supports having an else statement associated with a loop statement.

- If the **else** statement is used with a **for** loop, the **else** block is executed only if for loops terminates normally (and not by encountering break statement).

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a **for** statement that searches for even number in given list.

```
#!/usr/bin/python3
numbers=[11,33,55,39,55,75,37,21,23,41,13]
for num in numbers:
    if num%2==0:
        print ('the list contains an even number')
        break
else:
    print ('the list doesnot contain even number')
```

When the above code is executed, it produces the following result-

```
the list does not contain even number
```

Nested loops

Python programming language allows the use of one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows-

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example a **for** loop can be inside a while loop or vice versa.

Example

The following program uses a nested-for loop to display multiplication tables from 1-10.

```
#!/usr/bin/python3
import sys
```

```
for i in range(1,11):
    for j in range(1,11):
        k=i*j
        print (k, end=' ')
    print()
```

The print() function inner loop has **end=' '** which appends a space instead of default newline. Hence, the numbers will appear in one row.

Last print() will be executed at the end of inner for loop.

When the above code is executed, it produces the following result –

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Loop Control Statements

The Loop control statements change the execution from its normal sequence. When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Let us go through the loop control statements briefly.

break statement

The **break** statement is used for premature termination of the current loop. After abandoning the loop, execution at the next statement is resumed, just like the traditional break statement in C.

The most common use of break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

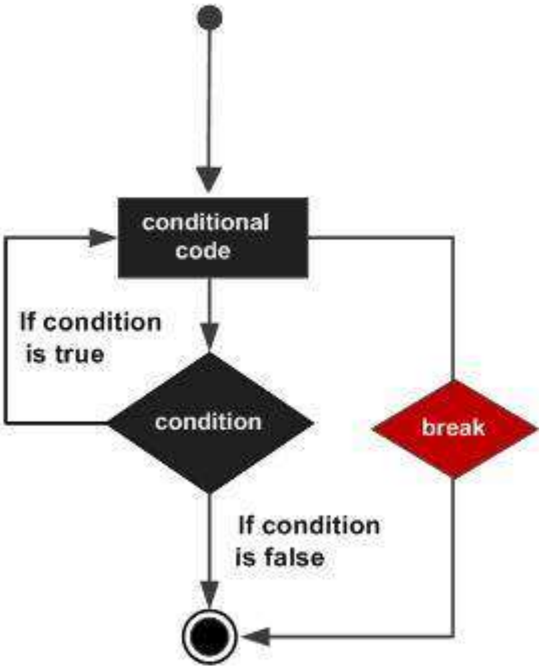
If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of the code after the block.

Syntax

The syntax for a **break** statement in Python is as follows-

```
break
```

Flow Diagram



Example

```
#!/usr/bin/python3
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   print ('Current variable value :', var)
   var = var -1
   if var == 5:
      break

print ("Good bye!")
```

When the above code is executed, it produces the following result-

```
Current Letter : P
Current Letter : y
Current Letter : t
```

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

The following program demonstrates the use of break in a for loop iterating over a list. User inputs a number, which is searched in the list. If it is found, then the loop terminates with the 'found' message.

```
#!/usr/bin/python3
no=int(input('any number: '))
numbers=[11,33,55,39,55,75,37,21,23,41,13]
for num in numbers:
    if num==no:
        print ('number found in list')
        break
else:
    print ('number not found in list')
```

The above program will produce the following output-

```
any number: 33
number found in list
any number: 5
number not found in list
```

continue Statement

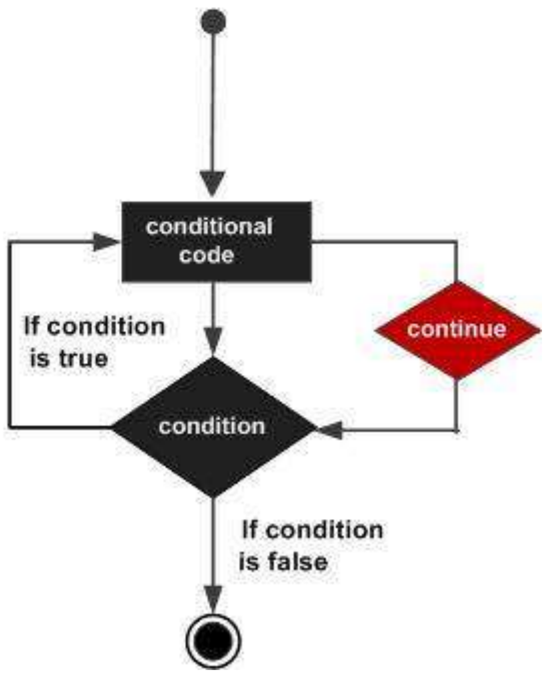
The **continue** statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.

The **continue** statement can be used in both *while* and *for* loops.

Syntax

```
continue
```

Flow Diagram



Example

```
#!/usr/bin/python3

for letter in 'Python':     # First Example
   if letter == 'h':
      continue
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   var = var -1
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!")
```

When the above code is executed, it produces the following result-

```
Current Letter : P
```

```
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs).

Syntax

```
pass
```

Example

```
#!/usr/bin/python3

for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)

print ("Good bye!")
```

When the above code is executed, it produces the following result-

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

Iterator and Generator

Iterator is an object, which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation. In Python, an iterator object implements two methods, **iter()** and **next().**

String, List or Tuple objects can be used to create an Iterator.

```
list=[1,2,3,4]
it = iter(list) # this builds an iterator object
print (next(it)) #prints next available element in iterator
Iterator object can be traversed using regular for statement
!usr/bin/python3
for x in it:
   print (x, end=" ")
or using next() function
while True:
   try:
      print (next(it))
   except StopIteration:
      sys.exit() #you have to import sys module for this
```

A **generator** is a function that produces or yields a sequence of values using yield method.

When a generator function is called, it returns a generator object without even beginning execution of the function. When the next() method is called for the first time, the function starts executing, until it reaches the yield statement, which returns the yielded value. The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.

The following example defines a generator, which generates an iterator for all the Fibonacci numbers.

```
!usr/bin/python3
```

```
import sys
def fibonacci(n): #generator function
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n):
            return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(5) #f is iterator object

while True:
   try:
      print (next(f), end=" ")
   except StopIteration:
      sys.exit()
```

9. Python 3 – Numbers

Number data types store numeric values. They are immutable data types. This means, changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example-

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the **del** statement. The syntax of the **del** statement is –

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the **del** statement. For example-

```
del var
del var_a, var_b
```

Python supports different numerical types-

- **int (signed integers)**: They are often called just integers or **ints**. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no **'long integer'** in Python 3 anymore.

- **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and the fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).

- **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

It is possible to represent an integer in hexa-decimal or octal form.

```
>>> number = 0xA0F #Hexa-decimal
>>> number
2575


>>> number=0o37 #Octal
>>> number
```

```
31
```

Examples

Here are some examples of numbers.

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

A complex number consists of an ordered pair of real floating-point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. Sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions.

Mathematical Functions

Python includes the following functions that perform mathematical calculations.

Function	Returns (Description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x.
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y. **Deprecated** in Python 3; Instead use **return (x>y)-(x<y).**
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x.
log(x)	The natural logarithm of x, for x> 0.
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity.
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity.
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
round(x [,n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
sqrt(x)	The square root of x for x > 0.

Let us learn about these functions in detail.

Number abs() Method

Description

The **abs()** method returns the absolute value of x i.e. the positive distance between x and zero.

Syntax

Following is the syntax for abs() method-

```
abs( x )
```

Parameters

x - This is a numeric expression.

Return Value

This method returns the absolute value of x.

Example

The following example shows the usage of the abs() method.

```
#!/usr/bin/python3
print ("abs(-45) : ", abs(-45))
print ("abs(100.12) : ", abs(100.12))
```

When we run the above program, it produces the following result-

```
abs(-45) :  45
abs(100.12) :  100.12
```

Number ceil() Method

Description

The **ceil()** method returns the ceiling value of x i.e. the smallest integer not less than x.

Syntax

Following is the syntax for the **ceil()** method-

```
import math
math.ceil( x )
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the smallest integer not less than x.

Example

The following example shows the usage of the ceil() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.ceil(-45.17) : ", math.ceil(-45.17))
print ("math.ceil(100.12) : ", math.ceil(100.12))
print ("math.ceil(100.72) : ", math.ceil(100.72))
print ("math.ceil(math.pi) : ", math.ceil(math.pi))
```

When we run the above program, it produces the following result-

```
math.ceil(-45.17) :  -45
math.ceil(100.12) :  101
math.ceil(100.72) :  101
math.ceil(math.pi) :  4
```

Number exp() Method

Description

The **exp()** method returns exponential of x: e^x.

Syntax

Following is the syntax for the exp() method-

```
import math
math.exp( x )
```

Note: This function is not accessible directly. Therefore, we need to import the math module and then we need to call this function using the math static object.

Parameters

X - This is a numeric expression.

Return Value

This method returns exponential of x: ex.

Example

The following example shows the usage of exp() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.exp(-45.17) : ", math.exp(-45.17))
print ("math.exp(100.12) : ", math.exp(100.12))
print ("math.exp(100.72) : ", math.exp(100.72))
print ("math.exp(math.pi) : ", math.exp(math.pi))
```

When we run the above program, it produces the following result-

```
math.exp(-45.17) :  2.4150062132629406e-20
math.exp(100.12) :  3.0308436140742566e+43
math.exp(100.72) :  5.522557130248187e+43
math.exp(math.pi) :  23.140692632779267
```

Number fabs() Method

Description

The **fabs()** method returns the absolute value of x. Although similar to the abs() function, there are differences between the two functions. They are-

- abs() is a built in function whereas fabs() is defined in math module.
- fabs() function works only on float and integer whereas abs() works with complex number also.

Syntax

Following is the syntax for the fabs() method-

```
import math
math.fabs( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric value.

Return Value

This method returns the absolute value of x.

Example

The following example shows the usage of the fabs() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.fabs(-45.17) : ", math.fabs(-45.17))
print ("math.fabs(100.12) : ", math.fabs(100.12))
print ("math.fabs(100.72) : ", math.fabs(100.72))
print ("math.fabs(math.pi) : ", math.fabs(math.pi))
```

When we run the above program, it produces following result-

```
math.fabs(-45.17) :  45.17
math.fabs(100) :  100.0
math.fabs(100.72) :  100.72
math.fabs(math.pi) :  3.141592653589793
```

Number floor() Method

Description

The **floor()** method returns the floor of **x** i.e. the largest integer not greater than x.

Syntax

Following is the syntax for the **floor()** method-

```
import math
math.floor( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the largest integer not greater than x.

Example

The following example shows the usage of the floor() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.floor(-45.17) : ", math.floor(-45.17))
print ("math.floor(100.12) : ", math.floor(100.12))
print ("math.floor(100.72) : ", math.floor(100.72))
print ("math.floor(math.pi) : ", math.floor(math.pi))
```

When we run the above program, it produces the following result-

```
math.floor(-45.17) :  -46
math.floor(100.12) :  100
math.floor(100.72) :  100
math.floor(math.pi) :  3
```

Number log() Method

Description

The **log()** method returns the natural logarithm of x, for x > 0.

Syntax

Following is the syntax for the **log()** method-

```
import math
math.log( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns natural logarithm of x, for x > 0.

Example

The following example shows the usage of the log() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.log(100.12) : ", math.log(100.12))
print ("math.log(100.72) : ", math.log(100.72))
print ("math.log(math.pi) : ", math.log(math.pi))
```

When we run the above program, it produces the following result-

```
math.log(100.12) :  4.6063694665635735
math.log(100.72) :  4.612344389736092
math.log(math.pi) :  1.1447298858494002
```

Number log10() Method

Description

The **log10()** method returns base-10 logarithm of x for x > 0.

Syntax

Following is the syntax for **log10()** method-

```
import math
math.log10( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the base-10 logarithm of x for x > 0.

Example

The following example shows the usage of the **log10()** method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.log10(100.12) : ", math.log10(100.12))
print ("math.log10(100.72) : ", math.log10(100.72))
print ("math.log10(119) : ", math.log10(119))
print ("math.log10(math.pi) : ", math.log10(math.pi))
```

When we run the above program, it produces the following result-

```
math.log10(100.12) :  2.0005208409361854
math.log10(100.72) :  2.003115717099806
math.log10(119) :  2.0755469613925306
math.log10(math.pi) :  0.49714987269413385
```

Number max() Method

Description

The **max()** method returns the largest of its arguments i.e. the value closest to positive infinity.

Syntax

Following is the syntax for **max()** method-

```
max( x, y, z, .... )
```

Parameters

- **x** - This is a numeric expression.
- **y** - This is also a numeric expression.
- **z** - This is also a numeric expression.

Return Value

This method returns the largest of its arguments.

Example

The following example shows the usage of the max() method.

```
#!/usr/bin/python3
print ("max(80, 100, 1000) : ", max(80, 100, 1000))
print ("max(-20, 100, 400) : ", max(-20, 100, 400))
print ("max(-80, -20, -10) : ", max(-80, -20, -10))
print ("max(0, 100, -400) : ", max(0, 100, -400))
```

When we run the above program, it produces the following result-

```
max(80, 100, 1000) :  1000
max(-20, 100, 400) :  400
max(-80, -20, -10) :  -10
```

```
max(0, 100, -400) :  100
```

Number min() Method

Description

The method **min()** returns the smallest of its arguments i.e. the value closest to negative infinity.

Syntax

Following is the syntax for the **min()** method-

```
min( x, y, z, .... )
```

Parameters

- **x** - This is a numeric expression.
- **y** - This is also a numeric expression.
- **z** - This is also a numeric expression.

Return Value

This method returns the smallest of its arguments.

Example

The following example shows the usage of the **min()** method.

```
#!/usr/bin/python3
print ("min(80, 100, 1000) : ", min(80, 100, 1000))
print ("min(-20, 100, 400) : ", min(-20, 100, 400))
print ("min(-80, -20, -10) : ", min(-80, -20, -10))
print ("min(0, 100, -400) : ", min(0, 100, -400))
```

When we run the above program, it produces the following result-

```
min(80, 100, 1000) :  80
min(-20, 100, 400) :  -20
min(-80, -20, -10) :  -80
min(0, 100, -400) :  -400
```

Number modf() Method

Description

The **modf()** method returns the fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.

Syntax

Following is the syntax for the **modf()** method-

```
import math
math.modf( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the fractional and integer parts of x in a two-item tuple. Both the parts have the same sign as x. The integer part is returned as a float.

Example

The following example shows the usage of the **modf()** method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.modf(100.12) : ", math.modf(100.12))
print ("math.modf(100.72) : ", math.modf(100.72))
print ("math.modf(119) : ", math.modf(119))
print ("math.modf(math.pi) : ", math.modf(math.pi))
```

When we run the above program, it produces the following result-

```
math.modf(100.12) :  (0.12000000000000455, 100.0)
math.modf(100.72) :  (0.7199999999999989, 100.0)
math.modf(119) :  (0.0, 119.0)
math.modf(math.pi) :  (0.14159265358979312, 3.0)
```

Number pow() Method

Return Value

This method returns the value of x^y.

Example

The following example shows the usage of the **pow()** method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.pow(100, 2) : ", math.pow(100, 2))
print ("math.pow(100, -2) : ", math.pow(100, -2))
print ("math.pow(2, 4) : ", math.pow(2, 4))
print ("math.pow(3, 0) : ", math.pow(3, 0))
```

When we run the above program, it produces the following result-

```
math.pow(100, 2) :  10000.0
math.pow(100, -2) :  0.0001
math.pow(2, 4) :  16.0
math.pow(3, 0) :  1.0
```

Number round() Method

Description

round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.

Syntax

Following is the syntax for the round() method-

```
round( x [, n]  )
```

Parameters

- **x** - This is a numeric expression.
- **n** - Represents number of digits from decimal point up to which x is to be rounded. Default is 0.

Return Value

This method returns x rounded to n digits from the decimal point.

Example

The following example shows the usage of **round()** method.

```
#!/usr/bin/python3
print ("round(70.23456) : ", round(70.23456))
print ("round(56.659,1) : ", round(56.659,1))
print ("round(80.264, 2) : ", round(80.264, 2))
print ("round(100.000056, 3) : ", round(100.000056, 3))
print ("round(-100.000056, 3) : ", round(-100.000056, 3))
```

When we run the above program, it produces the following result-

```
round(70.23456) :  70
round(56.659,1) :  56.7
round(80.264, 2) :  80.26
round(100.000056, 3) :  100.0
round(-100.000056, 3) :  -100.0
```

Number sqrt() Method

Description

The **sqrt()** method returns the square root of x for x > 0.

Syntax

Following is the syntax for **sqrt()** method-

```
import math
math.sqrt( x )
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns square root of x for x > 0.

Example

The following example shows the usage of sqrt() method.

```
#!/usr/bin/python3
import math   # This will import math module
print ("math.sqrt(100) : ", math.sqrt(100))
print ("math.sqrt(7) : ", math.sqrt(7))
print ("math.sqrt(math.pi) : ", math.sqrt(math.pi))
```

When we run the above program, it produces the following result-

```
math.sqrt(100) :  10.0
math.sqrt(7) :  2.6457513110645907
math.sqrt(math.pi) :  1.7724538509055159
```

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes the following functions that are commonly used.

Function	Description
choice(seq)	A random item from a list, tuple, or string.
randrange ([start,] stop [,step])	A randomly selected element from range(start, stop, step).
random()	A random float r, such that 0 is less than or equal to r and r is less than 1.
seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
shuffle(lst)	Randomizes the items of a list in place. Returns None.
uniform(x, y)	A random float r, such that x is less than or equal to r and r is less than y.

Number choice() Method

Description

The **choice()** method returns a random item from a list, tuple, or string.

Syntax

Following is the syntax for **choice()** method-

```
choice( seq )
```

Note: This function is not accessible directly, so we need to import the random module and then we need to call this function using the random static object.

Parameters

seq - This could be a list, tuple, or string...

Return Value

This method returns a random item.

Example

The following example shows the usage of the **choice()** method.

```
#!/usr/bin/python3
import random
print ("returns a random number from range(100) : ",random.choice(range(100)))
print ("returns random element from list [1, 2, 3, 5, 9]) : ", random.choice([1, 2, 3, 5, 9]))
print ("returns random character from string 'Hello World' : ", random.choice('Hello World'))
```

When we run the above program, it produces a result similar to the following-

```
returns a random number from range(100) :  19
returns random element from list [1, 2, 3, 5, 9]) :  9
returns random character from string 'Hello World' :  r
```

Number randrange() Method

Description

The **randrange()** method returns a randomly selected element from range(start, stop, step).

Syntax

Following is the syntax for the **randrange()** method-

```
randrange ([start,] stop [,step])
```

Note: This function is not accessible directly, so we need to import the random module and then we need to call this function using the random static object.

Parameters

- **start** - Start point of the range. This would be included in the range. Default is 0.
- **stop** - Stop point of the range. This would be excluded from the range.
- **step** - Value with which number is incremented. Default is 1.

Return Value

This method returns a random item from the given range.

Example

The following example shows the usage of the randrange() method.

```
#!/usr/bin/python3
import random
# randomly select an odd number between 1-100
print ("randrange(1,100, 2) : ", random.randrange(1, 100, 2))
# randomly select a number between 0-99
print ("randrange(100) : ", random.randrange(100))
```

When we run the above program, it produces the following result-

```
randrange(1,100, 2) :  83
randrange(100) :  93
```

Number random() Method

Description

The **random()** method returns a random floating point number in the range [0.0, 1.0].

Syntax

Following is the syntax for the random() method-

```
random ( )
```

Note: This function is not accessible directly, so we need to import the random module and then we need to call this function using the random static object.

Parameters

NA

Return Value

This method returns a random float r, such that 0.0 <= r <= 1.0

Example

The following example shows the usage of the **random()** method.

```
#!/usr/bin/python3
import random
# First random number
print ("random() : ", random.random())
# Second random number
print ("random() : ", random.random())
```

When we run the above program, it produces the following result-

```
random() :  0.281954791393
random() :  0.309090465205
```

Number seed() Method

Description

The **seed()** method initializes the basic random number generator. Call this function before calling any other random module function.

Syntax

Following is the syntax for the seed() method-

```
seed ([x], [y])
```

Note: This function initializes the basic random number generator.

Parameters

- **x -** This is the seed for the next random number. If omitted, then it takes system time to generate the next random number. If x is an int, it is used directly.
- **Y -** This is version number (default is 2). str, byte or byte array object gets converted in int. Version 1 used hash() of x.

Return Value

This method does not return any value.

Example

The following example shows the usage of the seed() method.

```
#!/usr/bin/python3
import random
random.seed()
print ("random number with default seed", random.random())
random.seed(10)
print ("random number with int seed", random.random())
random.seed("hello",2)
print ("random number with string seed", random.random())
```

When we run above program, it produces following result-

```
random number with default seed 0.2524977842762465
random number with int seed 0.5714025946899135
random number with string seed 0.3537754404730722
```

Number shuffle() Method

Description

The **shuffle()** method randomizes the items of a list in place.

Syntax

Following is the syntax for the **shuffle()** method-

```
shuffle (lst,[random])
```

Note: This function is not accessible directly, so we need to import the shuffle module and then we need to call this function using the random static object.

Parameters

- **lst** - This could be a list or tuple.
- **random** - This is an optional 0 argument function returning float between 0.0 - 1.0. Default is None.

Return Value

This method returns reshuffled list.

Example

The following example shows the usage of the shuffle() method.

```
#!/usr/bin/python3
import random
list = [20, 16, 10, 5];
random.shuffle(list)
print ("Reshuffled list : ",  list)
random.shuffle(list)
print ("Reshuffled list : ",  list)
```

When we run the above program, it produces the following result-

```
Reshuffled list :  [16, 5, 10, 20]
reshuffled list :  [20, 5, 10, 16]
```

Number uniform() Method

Description

The **uniform()** method returns a random float r, such that x is less than or equal to r and r is less than y.

Syntax

Following is the syntax for the **uniform()** method-

```
uniform(x, y)
```

Note: This function is not accessible directly, so we need to import the uniform module and then we need to call this function using the random static object.

Parameters

- **x** - Sets the lower limit of the random float.
- **y** - Sets the upper limit of the random float.

Return Value

This method returns a floating point number r such that x <=r < y.

Example

The following example shows the usage of the uniform() method.

```
#!/usr/bin/python3
import random
```

```
print ("Random Float uniform(5, 10) : ",  random.uniform(5, 10))
print ("Random Float uniform(7, 14) : ",  random.uniform(7, 14))
```

Let us run the above program. This will produce the following result-

```
Random Float uniform(5, 10) :  5.52615217015
Random Float uniform(7, 14) :  12.5326369199
```

Trigonometric Functions

Python includes the following functions that perform trigonometric calculations.

Function	Description
acos(x)	Return the arc cosine of x, in radians.
asin(x)	Return the arc sine of x, in radians.
atan(x)	Return the arc tangent of x, in radians.
atan2(y, x)	Return atan(y / x), in radians.
cos(x)	Return the cosine of x radians.
hypot(x, y)	Return the Euclidean norm, sqrt(x*x + y*y).
sin(x)	Return the sine of x radians.
tan(x)	Return the tangent of x radians.
degrees(x)	Converts angle x from radians to degrees.
radians(x)	Converts angle x from degrees to radians.

Number acos() Method

Description

The **acos()** method returns the arc cosine of x in radians.

Syntax

Following is the syntax for acos() method-

```
acos(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value in the range -1 to 1. If x is greater than 1 then it will generate 'math domain error'.

Return Value

This method returns arc cosine of x, in radians.

Example

The following example shows the usage of the acos() method.

```
#!/usr/bin/python3
import math
print ("acos(0.64) : ",  math.acos(0.64))
print ("acos(0) : ",  math.acos(0))
print ("acos(-1) : ",  math.acos(-1))
print ("acos(1) : ",  math.acos(1))
```

When we run the above program, it produces the following result-

```
acos(0.64) :  0.876298061168
acos(0) :  1.57079632679
acos(-1) :  3.14159265359
acos(1) :  0.0
```

Number asin() Method

Description

The **asin()** method returns the arc sine of x (in radians).

Syntax

Following is the syntax for the asin() method-

```
asin(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function usingthe math static object.

Parameters

x - This must be a numeric value in the range -1 to 1. If x is greater than 1 then it will generate 'math domain error'.

Return Value

This method returns arc sine of x, in radians.

Example

The following example shows the usage of the asin() method.

```
#!/usr/bin/python3
import math
print ("asin(0.64) : ",  math.asin(0.64))
print ("asin(0) : ",  math.asin(0))
print ("asin(-1) : ",  math.asin(-1))
print ("asin(1) : ",  math.asin(1))
```

When we run the above program, it produces the following result-

```
asin(0.64) :  0.694498265627
asin(0) :  0.0
asin(-1) :  -1.57079632679
asin(1) :  1.5707963267
```

Number atan() Method

Description

The atan() method returns the arc tangent of x, in radians.

Syntax

Following is the syntax for atan() method-

```
atan(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns arc tangent of x, in radians.

Example

The following example shows the usage of the atan() method.

```
#!/usr/bin/python3
import math
print ("atan(0.64) : ",  math.atan(0.64))
print ("atan(0) : ",  math.atan(0))
print ("atan(10) : ",  math.atan(10))
print ("atan(-1) : ",  math.atan(-1))
print ("atan(1) : ",  math.atan(1))
```

When we run the above program, it produces the following result-

```
atan(0.64) :  0.569313191101
atan(0) :  0.0
atan(10) :  1.4711276743
atan(-1) :  -0.785398163397
atan(1) :  0.785398163397
```

Number atan2() Method

Description

The **atan2()** method returns atan(y / x), in radians.

Syntax

Following is the syntax for atan2() method-

```
atan2(y, x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

- **y** - This must be a numeric value.
- **x** - This must be a numeric value.

Return Value

This method returns atan(y / x), in radians.

Example

The following example shows the usage of atan2() method.

```
#!/usr/bin/python3
import math
print ("atan2(-0.50,-0.50) : ",  math.atan2(-0.50,-0.50))
print ("atan2(0.50,0.50) : ",  math.atan2(0.50,0.50))
print ("atan2(5,5) : ",  math.atan2(5,5))
print ("atan2(-10,10) : ",  math.atan2(-10,10))
print ("atan2(10,20) : ",  math.atan2(10,20))
```

When we run the above program, it produces the following result-

```
atan2(-0.50,-0.50) :  -2.35619449019
atan2(0.50,0.50) :  0.785398163397
atan2(5,5) :  0.785398163397
atan2(-10,10) :  -0.785398163397
atan2(10,20) :  0.463647609001
```

Number cos() Method

Description

The **cos()** method returns the cosine of x radians.

Syntax

Following is the syntax for **cos()** method-

```
cos(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the cosine of the angle.

Example

The following example shows the usage of cos() method.

```
#!/usr/bin/python3
import math
print ("cos(3) : ",  math.cos(3))
print ("cos(-3) : ",  math.cos(-3))
print ("cos(0) : ",  math.cos(0))
print ("cos(math.pi) : ",  math.cos(math.pi))
print ("cos(2*math.pi) : ",  math.cos(2*math.pi))
```

When we run the above program, it produces the following result-

```
cos(3) :  -0.9899924966
cos(-3) :  -0.9899924966
cos(0) :  1.0
cos(math.pi) :  -1.0
cos(2*math.pi) :  1.0
```

Number hypot() Method

Description

The method hypot() return the Euclidean norm, sqrt(x*x + y*y). This is length of vector from origin to point (x,y)

Syntax

Following is the syntax for hypot() method-

```
hypot(x, y)
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.

Parameters

- **x** - This must be a numeric value.
- **y** - This must be a numeric value.

Return Value

This method returns Euclidean norm, sqrt(x*x + y*y).

Example

The following example shows the usage of hypot() method.

```
#!/usr/bin/python3
import math
print ("hypot(3, 2) : ",  math.hypot(3, 2))
print ("hypot(-3, 3) : ",  math.hypot(-3, 3))
print ("hypot(0, 2) : ",  math.hypot(0, 2))
```

When we run the above program, it produces the following result-

```
hypot(3, 2) :  3.60555127546
hypot(-3, 3) :  4.24264068712
hypot(0, 2) :  2.0
```

Number sin() Method

Description

The **sin()** method returns the sine of x, in radians.

Syntax

Following is the syntax for sin() method-

```
sin(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the sine of the parameter x.

Example

The following example shows the usage of sin() method.

```
#!/usr/bin/python3
import math
print ("sin(3) : ",  math.sin(3))
print ("sin(-3) : ",  math.sin(-3))
print ("sin(0) : ",  math.sin(0))
```

```
print ("sin(math.pi) : ",  math.sin(math.pi))
print ("sin(math.pi/2) : ",  math.sin(math.pi/2))
```

When we run the above program, it produces the following result-

```
sin(3) :  0.14112000806
sin(-3) :  -0.14112000806
sin(0) :  0.0
sin(math.pi) :  1.22460635382e-16
sin(math.pi/2) :  1
```

Number tan() Method

Description

The **tan()** method returns the tangent of x radians.

Syntax

Following is the syntax for tan() method.

```
tan(x)
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the tangent of the parameter x.

Example

The following example shows the usage of tan() method.

```
#!/usr/bin/python3
import math
print ("(tan(3) : ",  math.tan(3))
print ("tan(-3) : ",  math.tan(-3))
print ("tan(0) : ",  math.tan(0))
print ("tan(math.pi) : ",  math.tan(math.pi))
print ("tan(math.pi/2) : ",  math.tan(math.pi/2))
```

```
print ("tan(math.pi/4) : ",  math.tan(math.pi/4))
```

When we run the above program, it produces the following result-

```
print ("(tan(3) : ",  math.tan(3))
print ("tan(-3) : ",  math.tan(-3))
print ("tan(0) : ",  math.tan(0))
print ("tan(math.pi) : ",  math.tan(math.pi))
print ("tan(math.pi/2) : ",  math.tan(math.pi/2))
print ("tan(math.pi/4) : ",  math.tan(math.pi/4))
```

Number degrees() Method

Description

The **degrees()** method converts angle x from radians to degrees..

Syntax

Following is the syntax for degrees() method-

```
degrees(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns the degree value of an angle.

Example

The following example shows the usage of degrees() method.

```
#!/usr/bin/python3
import math
print ("degrees(3) : ",  math.degrees(3))
print ("degrees(-3) : ",  math.degrees(-3))
print ("degrees(0) : ",  math.degrees(0))
print ("degrees(math.pi) : ",  math.degrees(math.pi))
print ("degrees(math.pi/2) : ",  math.degrees(math.pi/2))
```

```
print ("degrees(math.pi/4) : ",  math.degrees(math.pi/4))
```

When we run the above program, it produces the following result-

```
degrees(3) :  171.88733853924697
degrees(-3) :  -171.88733853924697
degrees(0) :  0.0
degrees(math.pi) :  180.0
degrees(math.pi/2) :  90.0
degrees(math.pi/4) :  45.0
```

Number radians() Method

Description

The **radians()** method converts angle x from degrees to radians.

Syntax

Following is the syntax for radians() method-

```
radians(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns radian value of an angle.

Example

The following example shows the usage of radians() method.

```
#!/usr/bin/python3
import math
print ("radians(3) : ",  math.radians(3))
print ("radians(-3) : ",  math.radians(-3))
print ("radians(0) : ",  math.radians(0))
print ("radians(math.pi) : ",  math.radians(math.pi))
print ("radians(math.pi/2) : ",  math.radians(math.pi/2))
```

```
print ("radians(math.pi/4) : ",  math.radians(math.pi/4))
```

When we run the above program, it produces the following result-

```
radians(3) :  0.0523598775598
radians(-3) :  -0.0523598775598
radians(0) :  0.0
radians(math.pi) :  0.0548311355616
radians(math.pi/2) :  0.0274155677808
radians(math.pi/4) :  0.0137077838904
```

Mathematical Constants

The module also defines two mathematical constants-

Constants	Description
pi	The mathematical constant pi.
e	The mathematical constant e.

10. Python 3 – Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example-

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example-

```
#!/usr/bin/python3
var1 = 'Hello World!'
var2 = "Python Programming"
print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

When the above code is executed, it produces the following result-

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example-

```
#!/usr/bin/python3
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

When the above code is executed, it produces the following result-

```
Updated String :-  Hello Python
```

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
a	0x07	Bell or alert
b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab

\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then-

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n

%	Format - Performs String formatting	See next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example –

```
#!/usr/bin/python3
print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

When the above code is executed, it produces the following result –

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with %-

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')

%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table-

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Triple Quotes

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python3

para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
print (para_str)
```

When the above code is executed, it produces the following result. Note how every single special character has been converted to its printed form, right down to the last NEWLINE at the end of the string between the "up." and closing triple quotes. Also note that NEWLINEs occur either with an explicit carriage return at the end of a line or its escape code (\n) –

```
this is a long string that is made up of
several lines and non-printable characters such as
TAB (    ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
 ], or just a NEWLINE within
the variable assignment will also show up.
```

Raw strings do not treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it-

```
#!/usr/bin/python3
print ('C:\\nowhere')
```

When the above code is executed, it produces the following result-

```
C:\nowhere
```

Now let us make use of raw string. We would put expression in **r'expression'** as follows-

```
#!/usr/bin/python3
```

```
print (r'C:\\nowhere')
```

When the above code is executed, it produces the following result-

```
C:\\nowhere
```

Unicode String

In Python 3, all strings are represented in Unicode. In Python 2 are stored internally as 8-bit ASCII, hence it is required to attach 'u' to make it Unicode. It is no longer necessary now.

Built-in String Methods

Python includes the following built-in methods to manipulate strings-

S. No.	Methods with Description
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with *fillchar* with the original string centered to a total of *width* columns.
3	**count(str, beg= 0,end=len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	**decode(encoding='UTF-8',errors='strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding='UTF-8',errors='strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg=0, end=len(string))**

	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize=8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg=0 end=len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg=0, end=len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if the string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.

15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string
20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.

24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	**rfind(str, beg=0,end=len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg=0, end=len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.

33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string
35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.

String capitalize() Method

It returns a copy of the string with only its first character capitalized.

Syntax

```
str.capitalize()
```

Parameters

NA

Return Value

string

Example

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("str.capitalize() : ", str.capitalize())
```

Result

```
str.capitalize() :  This is string example....wow!!!
```

String center() Method

The method center() returns centered in a string of length width. Padding is done using the specified fillchar. Default filler is a space.

Syntax

```
str.center(width[, fillchar])
```

Parameters

- width - This is the total width of the string.
- fillchar - This is the filler character.

Return Value

This method returns a string that is at least width characters wide, created by padding the string with the character fillchar (default is a space).

Example

The following example shows the usage of the center() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
```

```
print ("str.center(40, 'a') : ", str.center(40, 'a'))
```

Result

```
str.center(40, 'a') :  aaaathis is string example....wow!!!aaaa
```

String count() Method

Description

The **count()** method returns the number of occurrences of substring sub in the range [start, end]. Optional arguments start and end are interpreted as in slice notation.

Syntax

```
str.count(sub, start= 0,end=len(string))
```

Parameters

- **sub -** This is the substring to be searched.
- **start** - Search starts from this index. First character starts from 0 index. By default search starts from 0 index.
- **end** - Search ends from this index. First character starts from 0 index. By default search ends at the last index.

Return Value

Centered in a string of length width.

Example

```
#!/usr/bin/python3
str="this is string example....wow!!!"
sub='i'
print ("str.count('i') : ", str.count(sub))
sub='exam'
print ("str.count('exam', 10, 40) : ", str.count(sub,10,40))
```

Result

```
str.count('i') :  3
str.count('exam', 4, 40) :
```

String decode() Method

Description

The **decode()** method decodes the string using the codec registered for encoding. It defaults to the default string encoding.

Syntax

```
Str.decode(encoding='UTF-8',errors='strict')
```

Parameters

- **encoding** - This is the encodings to be used. For a list of all encoding schemes please visit: Standard Encodings.

- **errors** - This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register_error()..

Return Value

Decoded string.

Example

```
#!/usr/bin/python3
Str = "this is string example....wow!!!";
Str = Str.encode('base64','strict');
print "Encoded String: " + Str
print "Decoded String: " + Str.decode('base64','strict')
```

Result

```
Encoded String:  b'dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE='
Decoded String:  this is string example....wow!!!
```

String encode() Method

Description

The **encode()** method returns an encoded version of the string. Default encoding is the current default string encoding. The errors may be given to set a different error handling scheme.

Syntax

```
str.encode(encoding='UTF-8',errors='strict')
```

Parameters

- **encoding -** This is the encodings to be used. For a list of all encoding schemes please visit: Standard Encodings.

- **errors -** This may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a UnicodeError. Other possible values are 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other name registered via codecs.register_error().

Return Value

Decoded string.

Example

```
#!/usr/bin/python3
import base64
Str = "this is string example....wow!!!"
Str=base64.b64encode(Str.encode('utf-8',errors='strict'))
print ("Encoded String: " , Str)
```

Result

```
Encoded String:  b'dGhpcyBpcyBzdHJpbmcgZXhhbXBsZS4uLi53b3chISE='
```

String endswith() Method

Description

It returns True if the string ends with the specified suffix, otherwise return False optionally restricting the matching with the given indices start and end.

Syntax

```
str.endswith(suffix[, start[, end]])
```

Parameters

- **suffix -** This could be a string or could also be a tuple of suffixes to look for.

- **start** - The slice begins from here.

- **end** - The slice ends here.

Return Value

TRUE if the string ends with the specified suffix, otherwise FALSE.

Example

```
#!/usr/bin/python3
Str='this is string example....wow!!!'
suffix='!!'
print (Str.endswith(suffix))
print (Str.endswith(suffix,20))
suffix='exam'
print (Str.endswith(suffix))
print (Str.endswith(suffix, 0, 19))
```

Result

```
True
True
False
True
```

String expandtabs() Method

Description

The **expandtabs()** method returns a copy of the string in which the tab characters ie. '\t' are expanded using spaces, optionally using the given tabsize (default 8)..

Syntax

```
str.expandtabs(tabsize=8)
```

Parameters

tabsize - This specifies the number of characters to be replaced for a tab character '\t'.

Return Value

This method returns a copy of the string in which tab characters i.e., '\t' have been expanded using spaces.

Example

```
#!/usr/bin/python3
str = "this is\tstring example....wow!!!"
print ("Original string: " + str)
print ("Defualt exapanded tab: " +  str.expandtabs())
print ("Double exapanded tab: " +  str.expandtabs(16))
```

Result

```
Original string: this is        string example....wow!!!
Defualt exapanded tab:         this is string example....wow!!!
Double exapanded tab: this is         string example....wow!!!
```

String find() Method

Description

The find() method determines if the string str occurs in string, or in a substring of string if the starting index beg and ending index end are given.

Syntax

```
str.find(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.
- **beg** - This is the starting index, by default its 0.
- **end** - This is the ending index, by default its equal to the lenght of the string.

Return Value

Index if found and -1 otherwise.

Example

```
#!/usr/bin/python3
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.find(str2))
print (str1.find(str2, 10))
print (str1.find(str2, 40))
```

Result

```
15
15
-1
```

String index() Method

Description

The index() method determines if the string str occurs in string or in a substring of string, if the starting index beg and ending index end are given. This method is same as find(), but raises an exception if sub is not found.

Syntax

```
str.index(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.
- **beg** - This is the starting index, by default its 0.
- **end** - This is the ending index, by default its equal to the length of the string.

Return Value

Index if found otherwise raises an exception if str is not found.

Example

```
#!/usr/bin/python3
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.index(str2))
print (str1.index(str2, 10))
print (str1.index(str2, 40))
```

Result

```
15
```

```
15
Traceback (most recent call last):
  File "test.py", line 7, in
    print (str1.index(str2, 40))
ValueError: substring not found
shell returned 1
```

String isalnum() Method

Description

The **isalnum()** method checks whether the string consists of alphanumeric characters.

Syntax

Following is the syntax for isalnum() method-

```
str.isa1num()
```

Parameters

NA

Return Value

This method returns true if all the characters in the string are alphanumeric and there is at least one character, false otherwise.

Example

The following example shows the usage of isalnum() method.

```
#!/usr/bin/python3
str = "this2016"  # No space in this string
print (str.isalnum())
str = "this is string example....wow!!!"
print (str.isalnum())
```

When we run the above program, it produces the following result-

```
True
False
```

String isalpha() Method

Description

The **isalpha()** method checks whether the string consists of alphabetic characters only.

Syntax

Following is the syntax for islpha() method-

```
str.isalpha()
```

Parameters

NA

Return Value

This method returns true if all the characters in the string are alphabetic and there is at least one character, false otherwise.

Example

The following example shows the usage of isalpha() method.

```
#!/usr/bin/python3
str = "this";  # No space & digit in this string
print (str.isalpha())
str = "this is string example....wow!!!"
print (str.isalpha())
```

Result

```
True
False
```

String isdigit() Method

Description

The method isdigit() checks whether the string consists of digits only.

Syntax

Following is the syntax for isdigit() method-

```
str.isdigit()
```

Parameters

NA

Return Value

This method returns true if all characters in the string are digits and there is at least one character, false otherwise.

Example

The following example shows the usage of isdigit() method.

```
#!/usr/bin/python3
str = "123456";  # Only digit in this string
print (str.isdigit())
str = "this is string example....wow!!!"
print (str.isdigit())
```

Result

```
True
False
```

String islower() Method

Description

The **islower()** method checks whether all the case-based characters (letters) of the string are lowercase.

Syntax

Following is the syntax for islower() method-

```
str.islower()
```

Parameters

NA

Return Value

This method returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Example

The following example shows the usage of islower() method.

```
#!/usr/bin/python3
str = "THIS is string example....wow!!!"
print (str.islower())
str = "this is string example....wow!!!"
print (str.islower())
```

Result

```
False
True
```

String isnumeric() Method

Description

The **isnumeric()** method checks whether the string consists of only numeric characters. This method is present only on unicode objects.

Note: Unlike Python 2, all strings are represented in Unicode in Python 3. Given below is an example illustrating it.

Syntax

Following is the syntax for isnumeric() method-

```
str.isnumeric()
```

Parameters

NA

Return Value

This method returns true if all characters in the string are numeric, false otherwise.

Example

The following example shows the usage of isnumeric() method.

```
#!/usr/bin/python3
str = "this2016"
print (str.isnumeric())
str = "23443434"
```

```
print (str.isnumeric())
```

Result

```
False
True
```

String isspace() Method

Description

The **isspace()** method checks whether the string consists of whitespace..

Syntax

Following is the syntax for isspace() method-

```
str.isspace()
```

Parameters

NA

Return Value

This method returns true if there are only whitespace characters in the string and there is at least one character, false otherwise.

Example

The following example shows the usage of isspace() method.

```
#!/usr/bin/python3
str = "       "
print (str.isspace())
str = "This is string example....wow!!!"
print (str.isspace())
```

Result

```
True
False
```

String istitle() Method

Description

The **istitle()** method checks whether all the case-based characters in the string following non-casebased letters are uppercase and all other case-based characters are lowercase.

Syntax

Following is the syntax for istitle() method-

```
str.istitle()
```

Parameters

NA

Return Value

This method returns true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. It returns false otherwise.

Example

The following example shows the usage of istitle() method.

```
#!/usr/bin/python3
str = "This Is String Example...Wow!!!"
print (str.istitle())
str = "This is string example....wow!!!"
print (str.istitle())
```

Result

```
True
False
```

String isupper() Method

Description

The **isupper()** method checks whether all the case-based characters (letters) of the string are uppercase.

Syntax

Following is the syntax for isupper() method-

```
str.isupper()
```

Parameters

NA

Return Value

This method returns true if all the cased characters in the string are uppercase and there is at least one cased character, false otherwise.

Example

The following example shows the usage of isupper() method.

```
#!/usr/bin/python3
str = "THIS IS STRING EXAMPLE....WOW!!!"
print (str.isupper())
str = "THIS is string example....wow!!!"
print (str.isupper())
```

Result

```
True
False
```

String join() Method

Description

The **join()** method returns a string in which the string elements of sequence have been joined by str separator.

Syntax

Following is the syntax for join() method-

```
str.join(sequence)
```

Parameters

sequence - This is a sequence of the elements to be joined.

Return Value

This method returns a string, which is the concatenation of the strings in the sequence **seq**. The separator between elements is the string providing this method.

Example

The following example shows the usage of join() method.

```
#!/usr/bin/python3
s = "-"
seq = ("a", "b", "c") # This is sequence of strings.
print (s.join( seq ))
```

Result

```
a-b-c
```

String len() Method

Description

The **len()** method returns the length of the string.

Syntax

Following is the syntax for len() method –

```
len( str )
```

Parameters

NA

Return Value

This method returns the length of the string.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("Length of the string: ", len(str))
```

Result

```
Length of the string:  32
```

String ljust() Method

Description

The method ljust() returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for ljust() method –

```
str.ljust(width[, fillchar])
```

Parameters

- **width** - This is string length in total after padding.
- **fillchar** - This is filler character, default is a space.

Return Value

This method returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Example

The following example shows the usage of ljust() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print str.ljust(50, '*')
```

Result

```
this is string example....wow!!!******************
```

String lower() Method

Description

The method lower() returns a copy of the string in which all case-based characters have been lowercased.

Syntax

Following is the syntax for lower() method −

```
str.lower()
```

Parameters

NA

Return Value

This method returns a copy of the string in which all case-based characters have been lowercased.

Example

The following example shows the usage of lower() method.

```
#!/usr/bin/python3
str = "THIS IS STRING EXAMPLE....WOW!!!"
print (str.lower())
```

Result

```
this is string example....wow!!!
```

String lstrip() Method

Description

The **lstrip()** method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Syntax

Following is the syntax for lstrip() method-

```
str.lstrip([chars])
```

Parameters

chars - You can supply what chars have to be trimmed.

Return Value

This method returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

Example

The following example shows the usage of lstrip() method.

```
#!/usr/bin/python3
str = "     this is string example....wow!!!"
print (str.lstrip())
str = "*****this is string example....wow!!!*****"
print (str.lstrip('*'))
```

Result

```
this is string example....wow!!!
this is string example....wow!!!*****
```

String maketrans() Method

Description

The **maketrans()** method returns a translation table that maps each character in the intabstring into the character at the same position in the outtab string. Then this table is passed to the translate() function.

Note: Both intab and outtab must have the same length.

Syntax

Following is the syntax for maketrans() method-

```
str.maketrans(intab, outtab]);
```

Parameters

- **intab** - This is the string having actual characters.
- **outtab** - This is the string having corresponding mapping character.

Return Value

This method returns a translate table to be used translate() function.

Example

The following example shows the usage of maketrans() method. Under this, every vowel in a string is replaced by its vowel position –

```
#!/usr/bin/python3
intab = "aeiou"
```

```
outtab = "12345"
trantab = str.maketrans(intab, outtab)
str = "this is string example....wow!!!"
print (str.translate(trantab))
```

Result

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

String max() Method

Description

The **max()** method returns the max alphabetical character from the string str.

Syntax

Following is the syntax for max() method-

```
max(str)
```

Parameters

str - This is the string from which max alphabetical character needs to be returned.

Return Value

This method returns the max alphabetical character from the string str.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
str = "this is a string example....really!!!"
print ("Max character: " + max(str))
str = "this is a string example....wow!!!"
print ("Max character: " + max(str))
```

Result

```
Max character: y
Max character: x
```

String min() Method

Description

The **min()** method returns the min alphabetical character from the string str.

Syntax

Following is the syntax for min() method-

```
min(str)
```

Parameters

str - This is the string from which min alphabetical character needs to be returned.

Return Value

This method returns the max alphabetical character from the string str.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python3
str = "www.tutorialspoint.com"
print ("Min character: " + min(str))
str = "TUTORIALSPOINT"
print ("Min character: " + min(str))
```

Result

```
Min character: .
Min character: A
```

String replace() Method

Description

The **replace()** method returns a copy of the string in which the occurrences of old have been replaced with new, optionally restricting the number of replacements to max.

Syntax

Following is the syntax for replace() method-

```
str.replace(old, new[, max])
```

Parameters

- **old** - This is old substring to be replaced.
- **new** - This is new substring, which would replace old substring.
- **max** - If this optional argument max is given, only the first count occurrences are replaced.

Return Value

This method returns a copy of the string with all occurrences of substring old replaced by new. If the optional argument max is given, only the first count occurrences are replaced.

Example

The following example shows the usage of replace() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
```

Result

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

String rfind() Method

Description

The **rfind()** method returns the last index where the substring str is found, or -1 if no such index exists, optionally restricting the search to string[beg:end].

Syntax

Following is the syntax for rfind() method-

```
str.rfind(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.
- **beg** - This is the starting index, by default its 0.
- **end** - This is the ending index, by default its equal to the length of the string.

Return Value

This method returns last index if found and -1 otherwise.

Example

The following example shows the usage of rfind() method.

```
#!/usr/bin/python3
str1 = "this is really a string example....wow!!!"
str2 = "is"
print (str1.rfind(str2))
print (str1.rfind(str2, 0, 10))
print (str1.rfind(str2, 10, 0))
print (str1.find(str2))
print (str1.find(str2, 0, 10))
print (str1.find(str2, 10, 0))
```

Result

```
5
5
-1
2
2
-1
```

String rindex() Method

Description

The **rindex()** method returns the last index where the substring str is found, or raises an exception if no such index exists, optionally restricting the search to string[beg:end].

Syntax

Following is the syntax for rindex() method-

```
str.rindex(str, beg=0 end=len(string))
```

Parameters

- **str** - This specifies the string to be searched.

- **beg** - This is the starting index, by default its 0.
- **len** - This is ending index, by default its equal to the length of the string.

Return Value

This method returns last index if found otherwise raises an exception if str is not found.

Example

The following example shows the usage of rindex() method.

```
#!/usr/bin/python3
str1 = "this is really a string example....wow!!!"
str2 = "is"
print (str1.rindex(str2))
print (str1.rindex(str2,10))
```

Result

```
5
Traceback (most recent call last):
  File "test.py", line 5, in
    print (str1.rindex(str2,10))
ValueError: substring not found
```

String rjust() Method

Description

The **rjust()** method returns the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(s).

Syntax

Following is the syntax for rjust() method-

```
str.rjust(width[, fillchar])
```

Parameters

- **width** - This is the string length in total after padding.
- **fillchar** - This is the filler character, default is a space.

Return Value

This method returns the string right justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if the width is less than len(s).

Example

The following example shows the usage of rjust() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.rjust(50, '*'))
```

Result

```
******************this is string example....wow!!!
```

String rstrip() Method

Description

The **rstrip()** method returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

Syntax

Following is the syntax for rstrip() method-

```
str.rstrip([chars])
```

Parameters

chars - You can supply what chars have to be trimmed.

Return Value

This method returns a copy of the string in which all chars have been stripped from the end of the string (default whitespace characters).

Example

The following example shows the usage of rstrip() method.

```
#!/usr/bin/python3
str = "     this is string example....wow!!!     "
print (str.rstrip())
str = "*****this is string example....wow!!!*****"
```

```
print (str.rstrip('*'))
```

Result

```
     this is string example....wow!!!
*****this is string example....wow!!!
```

String split() Method

Description

The **split()** method returns a list of all the words in the string, using str as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to num.

Syntax

Following is the syntax for split() method-

```
str.split(str="", num=string.count(str)).
```

Parameters

- **str** - This is any delimeter, by default it is space.
- **num** - this is number of lines to be made

Return Value

This method returns a list of lines.

Example

The following example shows the usage of split() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.split( ))
print (str.split('i',1))
print (str.split('w'))
```

Result

```
['this', 'is', 'string', 'example....wow!!!']
['th', 's is string example....wow!!!']
['this is string example....', 'o', '!!!']
```

String splitlines() Method

Description

The **splitlines()** method returns a list with all the lines in string, optionally including the line breaks (if num is supplied and is true).

Syntax

Following is the syntax for splitlines() method-

```
str.splitlines( num=string.count('\n'))
```

Parameters

num - This is any number, if present then it would be assumed that the line breaks need to be included in the lines.

Return Value

This method returns true if found matching with the string otherwise false.

Example

The following example shows the usage of splitlines() method.

```
#!/usr/bin/python3
str = "this is \nstring example....\nwow!!!"
print (str.splitlines( ))
```

Result

```
['this is ', 'string example....', 'wow!!!']
```

String startswith() Method

Description

The **startswith()** method checks whether the string starts with str, optionally restricting the matching with the given indices start and end.

Syntax

Following is the syntax for startswith() method-

```
str.startswith(str, beg=0,end=len(string));
```

Parameters

- **str** - This is the string to be checked.
- **beg** - This is the optional parameter to set start index of the matching boundary.
- **end** - This is the optional parameter to set start index of the matching boundary.

Return Value

This method returns true if found matching with the string otherwise false.

Example

The following example shows the usage of startswith() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.startswith( 'this' ))
print (str.startswith( 'string', 8 ))
print (str.startswith( 'this', 2, 4 ))
```

Result

```
True
True
False
```

String strip() Method

Description

The **strip()** method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

Syntax

Following is the syntax for strip() method –

```
str.strip([chars]);
```

Parameters

chars - The characters to be removed from beginning or end of the string.

Return Value

This method returns a copy of the string in which all the chars have been stripped from the beginning and the end of the string.

Example

The following example shows the usage of strip() method.

```
#!/usr/bin/python3
str = "*****this is string example....wow!!!*****"
print (str.strip( '*' ))
```

Result

```
this is string example....wow!!!
```

String swapcase() Method

Description

The **swapcase()** method returns a copy of the string in which all the case-based characters have had their case swapped.

Syntax

Following is the syntax for swapcase() method-

```
str.swapcase();
```

Parameters

NA

Return Value

This method returns a copy of the string in which all the case-based characters have had their case swapped.

Example

The following example shows the usage of swapcase() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.swapcase())
str = "This Is String Example....WOW!!!"
print (str.swapcase())
```

Result

```
THIS IS STRING EXAMPLE....WOW!!!
tHIS iS sTRING eXAMPLE....wow!!!
```

String title() Method

Description

The **title()** method returns a copy of the string in which first characters of all the words are capitalized.

Syntax

Following is the syntax for title() method-

```
str.title();
```

Parameters

NA

Return Value

This method returns a copy of the string in which first characters of all the words are capitalized.

Example

The following example shows the usage of title() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print (str.title())
```

Result

```
This Is String Example....Wow!!!
```

String translate() Method

Description

The method translate() returns a copy of the string in which all the characters have been translated using table (constructed with the maketrans() function in the string module), optionally deleting all characters found in the string deletechars.

Syntax

Following is the syntax for translate() method-

```
str.translate(table[, deletechars]);
```

Parameters

- **table** - You can use the maketrans() helper function in the string module to create a translation table.
- **deletechars** - The list of characters to be removed from the source string.

Return Value

This method returns a translated copy of the string.

Example

The following example shows the usage of translate() method. Under this, every vowel in a string is replaced by its vowel position.

```
#!/usr/bin/python3
from string import maketrans   # Required to call maketrans function.
intab = "aeiou"
outtab = "12345"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!";
print (str.translate(trantab))
```

Result

```
th3s 3s str3ng 2x1mpl2....w4w!!!
```

Following is the example to delete 'x' and 'm' characters from the string-

```
#!/usr/bin/python3
from string import maketrans   # Required to call maketrans function.
intab = "aeiouxm"
outtab = "1234512"
trantab = maketrans(intab, outtab)
str = "this is string example....wow!!!";
print (str.translate(trantab))
```

Result

```
th3s 3s str3ng 21pl2....w4w!!!
```

String upper() Method

Description

The **upper()** method returns a copy of the string in which all case-based characters have been uppercased.

Syntax

Following is the syntax for upper() method –

```
str.upper()
```

Parameters

NA

Return Value

This method returns a copy of the string in which all case-based characters have been uppercased.

Example

The following example shows the usage of upper() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("str.upper : ",str.upper())
```

Result

```
str.upper :  THIS IS STRING EXAMPLE....WOW!!!
```

String zfill() Method

Description

The **zfill()** method pads string on the left with zeros to fill width.

Syntax

Following is the syntax for zfill() method-

```
str.zfill(width)
```

Parameters

width - This is final width of the string. This is the width which we would get after filling zeros.

Return Value

This method returns padded string.

Example

The following example shows the usage of zfill() method.

```
#!/usr/bin/python3
str = "this is string example....wow!!!"
print ("str.zfill : ",str.zfill(40))
print ("str.zfill : ",str.zfill(50))
```

Result

```
str.zfill :  00000000this is string example....wow!!!
str.zfill :  000000000000000000this is string example....wow!!!
```

String isdecimal() Method

Description

The **isdecimal()** method checks whether the string consists of only decimal characters. This method are present only on unicode objects.

Note: Unlike in Python 2, all strings are represented as Unicode in Python 3. Given Below is an example illustrating it.

Syntax

Following is the syntax for isdecimal() method-

```
str.isdecimal()
```

Parameters

NA

Return Value

This method returns true if all the characters in the string are decimal, false otherwise.

Example

The following example shows the usage of isdecimal() method.

```
#!/usr/bin/python3
str = "this2016"
print (str.isdecimal())
str = "23443434"
print (str.isdecimal())
```

Result

```
False
True
```

11. Python 3 – Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all the sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

The list is the most versatile datatype available in Python, which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that the items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example-

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"];
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example-

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5, 6, 7 ]
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result –

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example-

```
#!/usr/bin/python3
list = ['physics', 'chemistry', 1997, 2000]
print ("Value available at index 2 : ", list[2])
list[2] = 2001
print ("New value available at index 2 : ", list[2])
```

Note: The append() method is discussed in the subsequent section.

When the above code is executed, it produces the following result –

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting. You can use the remove() method if you do not know exactly which items to delete. For example-

```
#!/usr/bin/python3
list = ['physics', 'chemistry', 1997, 2000]
print (list)
del list[2]
print ("After deleting value at index 2 : ", list)
```

When the above code is executed, it produces the following result-

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :  ['physics', 'chemistry', 2000]
Note: remove() method is discussed in subsequent section.
```

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end=' ')	1 2 3	Iteration

Indexing, Slicing and Matrixes

Since lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming the following input-

```
L=['C++'', 'Java', 'Python']
```

Python Expression	Results	Description
L[2]	'Python'	Offsets start at zero
L[-2]	'Java'	Negative: count from the right
L[1:]	['Java', 'Python']	Slicing fetches sections

Built-in List Functions & Methods

Python includes the following list functions-

SN	Function with Description
1	**cmp(list1, list2)** No longer available in Python 3.

2	**len(list)** Gives the total length of the list.
3	**max(list)** Returns item from the list with max value.
4	**min(list)** Returns item from the list with min value.
5	**list(seq)** Converts a tuple into list.

Let us understand the use of these functions.

List len() Method

Description

The **len()** method returns the number of elements in the list.

Syntax

Following is the syntax for len() method-

```
len(list)
```

Parameters

list - This is a list for which, number of elements are to be counted.

Return Value

This method returns the number of elements in the list.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
print (len(list1))
list2=list(range(5)) #creates list of numbers between 0-4
print (len(list2))
```

When we run above program, it produces following result-

```
3
5
```

List max() Method

Description

The **max()** method returns the elements from the list with maximum value.

Syntax

Following is the syntax for max() method-

```
max(list)
```

Parameters

list - This is a list from which max valued element are to be returned.

Return Value

This method returns the elements from the list with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("Max value element : ", max(list1))
print ("Max value element : ", max(list2))
```

When we run above program, it produces following result-

```
Max value element :  Python
Max value element :  700
```

List min() Method

Description

The method min() returns the elements from the list with minimum value.

Syntax

Following is the syntax for min() method-

```
min(list)
```

Parameters

list - This is a list from which min valued element is to be returned.

Return Value

This method returns the elements from the list with minimum value.

Example

```
The following example shows the usage of min() method.
#!/usr/bin/python3
list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
print ("min value element : ", min(list1))
print ("min value element : ", min(list2))
```

When we run above program, it produces following result-

```
min value element :  C++
min value element :  200
```

List list() Method

Description

The **list()** method takes sequence types and converts them to lists. This is used to convert a given tuple into list.

Note: Tuple are very similar to lists with only difference that element values of a tuple can not be changed and tuple elements are put between parentheses instead of square bracket. This function also converts characters in a string into a list.

Syntax

Following is the syntax for list() method-

```
list( seq )
```

Parameters

seq - This is a tuple or string to be converted into list.

Return Value

This method returns the list.

Example

The following example shows the usage of list() method.

```
#!/usr/bin/python3
aTuple = (123, 'C++', 'Java', 'Python')
list1 = list(aTuple)
print ("List elements : ", list1)
str="Hello World"
list2=list(str)
print ("List elements : ", list2)
```

When we run above program, it produces following result-

```
List elements :  [123, 'C++', 'Java', 'Python']
List elements :  ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

Python includes the following list methods-

SN	Methods with Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index

6	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

List append() Method

Description

The **append()** method appends a passed obj into the existing list.

Syntax

Following is the syntax for append() method-

```
list.append(obj)
```

Parameters

obj - This is the object to be appended in the list.

Return Value

This method does not return any value but updates existing list.

Example

The following example shows the usage of append() method.

```
#!/usr/bin/python3
list1 = ['C++', 'Java', 'Python']
list1.append('C#')
print ("updated list : ", list1)
```

When we run the above program, it produces the following result-

```
updated list :  ['C++', 'Java', 'Python', 'C#']
```

List count() Method

Description

The **count()** method returns count of how many times obj occurs in list.

Syntax

Following is the syntax for count() method-

```
list.count(obj)
```

Parameters

obj - This is the object to be counted in the list.

Return Value

This method returns count of how many times obj occurs in list.

Example

The following example shows the usage of count() method.

```
#!/usr/bin/python3
aList = [123, 'xyz', 'zara', 'abc', 123];
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

When we run the above program, it produces the following result-

```
Count for 123 :  2
Count for zara :  1
```

List extend() Method

Description

The **extend()** method appends the contents of seq to list.

Syntax

Following is the syntax for extend() method-

```
list.extend(seq)
```

Parameters

seq - This is the list of elements

Return Value

This method does not return any value but adds the content to an existing list.

Example

The following example shows the usage of extend() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
list2=list(range(5)) #creates list of numbers between 0-4
list1.extend('Extended List :', list2)
print (list1)
```

When we run the above program, it produces the following result-

```
Extended List :  ['physics', 'chemistry', 'maths', 0, 1, 2, 3, 4]
```

List index() Method

Description

The **index()** method returns the lowest index in list that obj appears.

Syntax

Following is the syntax for index() method-

```
list.index(obj)
```

Parameters

obj - This is the object to be find out.

Return Value

This method returns index of the found object otherwise raises an exception indicating that the value is not found.

Example

The following example shows the usage of index() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
print ('Index of chemistry', list1.index('chemistry'))
print ('Index of C#', list1.index('C#'))
```

When we run the above program, it produces the following result-

```
Index of chemistry 1
Traceback (most recent call last):
  File "test.py", line 3, in
    print ('Index of C#', list1.index('C#'))
ValueError: 'C#' is not in list
```

List insert() Method

Description

The **insert() method** inserts object obj into list at offset index.

Syntax

Following is the syntax for insert() method-

```
list.insert(index, obj)
```

Parameters

- **index** - This is the Index where the object obj need to be inserted.
- **obj** - This is the Object to be inserted into the given list.

Return Value

This method does not return any value but it inserts the given element at the given index.

Example

The following example shows the usage of insert() method.

```
#!/usr/bin/python3
list1 = ['physics', 'chemistry', 'maths']
list1.insert(1, 'Biology')
print ('Final list : ', list1)
```

When we run the above program, it produces the following result-

```
Final list :  ['physics', 'Biology', 'chemistry', 'maths']
```

List pop() Method

Description

The **pop()** method removes and returns last object or obj from the list.

Syntax

Following is the syntax for pop() method-

```
list.pop(obj=list[-1])
```

Parameters

obj - This is an optional parameter, index of the object to be removed from the list.

Return Value

This method returns the removed object from the list.

Example

The following example shows the usage of pop() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.pop()
print ("list now : ", list1)
list1.pop(1)
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :  ['physics', 'Biology', 'chemistry']
list now :  ['physics', 'chemistry']
```

List remove() Method

Parameters

obj - This is the object to be removed from the list.

Return Value

This method does not return any value but removes the given object from the list.

Example

The following example shows the usage of remove() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.remove('Biology')
print ("list now : ", list1)
list1.remove('maths')
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :  ['physics', 'chemistry', 'maths']
list now :  ['physics', 'chemistry']
```

List reverse() Method

Description

The **reverse()** method reverses objects of list in place.

Syntax

Following is the syntax for reverse() method-

```
list.reverse()
```

Parameters

NA

Return Value

This method does not return any value but reverse the given object from the list.

Example

The following example shows the usage of reverse() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.reverse()
print ("list now : ", list1)
```

When we run above program, it produces following result-

```
list now :  ['maths', 'chemistry', 'Biology', 'physics']
```

List sort() Method

Description

The **sort()** method sorts objects of list, use compare function if given.

Syntax

Following is the syntax for sort() method-

```
list.sort([func])
```

Parameters

NA

Return Value

This method does not return any value but reverses the given object from the list.

Example

The following example shows the usage of sort() method.

```
#!/usr/bin/python3
list1 = ['physics', 'Biology', 'chemistry', 'maths']
list1.sort()
print ("list now : ", list1)
```

When we run the above program, it produces the following result-

```
list now :  ['Biology', 'chemistry', 'maths', 'physics']
```

12. Python 3 – Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also. For example-

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

The empty tuple is written as two parentheses containing nothing.

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value.

```
tup1 = (50,)
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain the value available at that index. For example-

```
#!/usr/bin/python3
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7 )
print ("tup1[0]: ", tup1[0])
print ("tup2[1:5]: ", tup2[1:5])
```

When the above code is executed, it produces the following result-

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

Updating Tuples

Tuples are immutable, which means you cannot update or change the values of tuple elements. You are able to take portions of the existing tuples to create new tuples as the following example demonstrates.

```
#!/usr/bin/python3

tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2
print (tup3)
```

When the above code is executed, it produces the following result-

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example-

```
#!/usr/bin/python3
tup = ('physics', 'chemistry', 1997, 2000);
print (tup)
del tup;
print "After deleting tup : "
print tup
```

This produces the following result.

Note: An exception is raised. This is because after **del tup,** tuple does not exist any more.

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
```

```
NameError: name 'tup' is not defined
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the previous chapter.

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1,2,3) : print (x, end=' ')	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Since tuples are sequences, indexing and slicing work the same way for tuples as they do for strings, assuming the following input-

```
T=('C++', 'Java', 'Python')
```

Python Expression	Results	Description
T[2]	'Python'	Offsets start at zero
T[-2]	'Java'	Negative: count from the right
T[1:]	('Java', 'Python')	Slicing fetches sections

No Enclosing Delimiters

No enclosing Delimiters is any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples.

Built-in Tuple Functions

Python includes the following tuple functions-

SN	Function with Description
1	cmp(tuple1, tuple2) No longer available in Python 3.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

Tuple len() Method

Description

The **len()** method returns the number of elements in the tuple.

Syntax

Following is the syntax for len() method-

```
len(tuple)
```

Parameters

tuple - This is a tuple for which number of elements to be counted.

Return Value

This method returns the number of elements in the tuple.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print ("First tuple length : ", len(tuple1))
print ("Second tuple length : ", len(tuple2))
```

When we run above program, it produces following result-

```
First tuple length :  3
Second tuple length :  2
```

Tuple max() Method

Description

The **max()** method returns the elements from the tuple with maximum value.

Syntax

Following is the syntax for max() method-

```
max(tuple)
```

Parameters

tuple - This is a tuple from which max valued element to be returned.

Return Value

This method returns the elements from the tuple with maximum value.

Example

The following example shows the usage of max() method.

```
#!/usr/bin/python3
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
print ("Max value element : ", max(tuple1))
print ("Max value element : ", max(tuple2))
```

When we run the above program, it produces the following result-

```
Max value element :  phy
Max value element :  700
```

Tuple min() Method

Description

The **min()** method returns the elements from the tuple with minimum value.

Syntax

Following is the syntax for min() method-

```
min(tuple)
```

Parameters

tuple - This is a tuple from which min valued element is to be returned.

Return Value

This method returns the elements from the tuple with minimum value.

Example

The following example shows the usage of min() method.

```
#!/usr/bin/python3
tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
print ("min value element : ", min(tuple1))
print ("min value element : ", min(tuple2))
```

When we run the above program, it produces the following result-

```
min value element :  bio
min value element :  200
```

Tuple tuple() Method

Description

The **tuple()** method converts a list of items into tuples.

Syntax

Following is the syntax for tuple() method-

```
tuple( seq )
```

Parameters

seq - This is a tuple to be converted into tuple.

Return Value

This method returns the tuple.

Example

The following example shows the usage of tuple() method.

```
#!/usr/bin/python3
list1= ['maths', 'che', 'phy', 'bio']
tuple1=tuple(list1)
print ("tuple elements : ", tuple1)
```

When we run the above program, it produces the following result-

```
tuple elements :  ('maths', 'che', 'phy', 'bio')
```

13. Python 3 – Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example.

```
#!/usr/bin/python3

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

When the above code is executed, it produces the following result-

```
dict['Name']:  Zara
dict['Age']:  7
```

If we attempt to access a data item with a key, which is not a part of the dictionary, we get an error as follows-

```
#!/usr/bin/python3

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice']
```

When the above code is executed, it produces the following result-

```
dict['Zara']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
#!/usr/bin/python3

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

When the above code is executed, it produces the following result-

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example-

```
#!/usr/bin/python3

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name'] # remove entry with key 'Name'
dict.clear()     # remove all entries in dict
del dict         # delete entire dictionary

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

This produces the following result.

Note: An exception is raised because after **del dict,** the dictionary does not exist anymore.

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
```

```
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note: The del() method is discussed in subsequent section.

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys-

(a) More than one entry per key is not allowed. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins. For example-

```
#!/usr/bin/python3

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}

print ("dict['Name']: ", dict['Name'])
```

When the above code is executed, it produces the following result-

```
dict['Name']:  Manni
```

(b) Keys must be immutable. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example-

```
#!/usr/bin/python3

dict = {['Name']: 'Zara', 'Age': 7}

print ("dict['Name']: ", dict['Name'])
```

When the above code is executed, it produces the following result-

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    dict = {['Name']: 'Zara', 'Age': 7}
TypeError: list objects are unhashable
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions-

SN	Functions with Description
1	**cmp(dict1, dict2)** No longer available in Python 3.
2	**len(dict)** Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	**str(dict)** Produces a printable string representation of a dictionary.
4	**type(variable)** Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Dictionary len() Method

DescriptionThe method len() **gives the total length of the dictionary. This would be equal to the number of items in the dictionary.**

Syntax

Following is the syntax for len() method-

```
len(dict)
```

Parameters

dict - This is the dictionary, whose length needs to be calculated.

Return Value

This method returns the length.

Example

The following example shows the usage of len() method.

```
#!/usr/bin/python3
```

```
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Length : %d" % len (dict))
```

When we run the above program, it produces the following result-

```
Length : 3
```

Dictionary str() Method

Description

The method **str()** produces a printable string representation of a dictionary.

Syntax

Following is the syntax for str() method –

```
str(dict)
```

Parameters

dict - This is the dictionary.

Return Value

This method returns string representation.

Example

The following example shows the usage of str() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Equivalent String : %s" % str (dict))
```

When we run the above program, it produces the following result-

```
Equivalent String : {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
```

Dictionary type() Method

Description

The method **type()** returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.

Syntax

Following is the syntax for type() method-

```
type(dict)
```

Parameters

dict - This is the dictionary.

Return Value

This method returns the type of the passed variable.

Example

The following example shows the usage of type() method.

```
#!/usr/bin/python3
dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
print ("Variable Type : %s" %  type (dict))
```

When we run the above program, it produces the following result-

```
Variable Type : <type 'dict'>
```

Python includes the following dictionary methods-

SN	Methods with Description
1	**dict.clear()** Removes all elements of dictionary *dict.*
2	**dict.copy()** Returns a shallow copy of dictionary *dict.*
3	**dict.fromkeys()** Create a new dictionary with keys from seq and values *set* to *value*.
4	**dict.get(key, default=None)** For *key* key, returns value or default if key not in dictionary.

5	**dict.has_key(key)** Removed, use the **in** operation instead.
6	**dict.items()** Returns a list of *dict*'s (key, value) tuple pairs.
7	**dict.keys()** Returns list of dictionary dict's keys.
8	**dict.setdefault(key, default=None)** Similar to get(), but will set dict[key]=default if *key* is not already in dict.
9	**dict.update(dict2)** Adds dictionary *dict2*'s key-values pairs to *dict.*
10	**dict.values()** Returns list of dictionary *dict*'s values.

Dictionary clear() Method

Description

The method **clear()** removes all items from the dictionary.

Syntax

Following is the syntax for clear() method-

```
dict.clear()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of clear() method.

```
#!/usr/bin/python3
```

```
dict = {'Name': 'Zara', 'Age': 7}
print ("Start Len : %d" %  len(dict))
dict.clear()
print ("End Len : %d" %  len(dict))
```

When we run the above program, it produces the following result-

```
Start Len : 2
End Len : 0
```

Dictionary copy() Method

Description

The method **copy()** returns a shallow copy of the dictionary.

Syntax

Following is the syntax for copy() method-

```
dict.copy()
```

Parameters

NA

Return Value

This method returns a shallow copy of the dictionary.

Example

The following example shows the usage of copy() method.

```
#!/usr/bin/python3
dict1 = {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
dict2 = dict1.copy()
print ("New Dictionary : ",dict2)
```

When we run the above program, it produces following result-

```
New dictionary :  {'Name': 'Manni', 'Age': 7, 'Class': 'First'}
```

Dictionary fromkeys() Method

Description

The method fromkeys() creates a new dictionary with keys from seq and values set to value.

Syntax

Following is the syntax for fromkeys() method-

```
dict.fromkeys(seq[, value]))
```

Parameters

- **seq** - This is the list of values which would be used for dictionary keys preparation.
- **value** - This is optional, if provided then value would be set to this value

Return Value

This method returns the list.

Example

The following example shows the usage of fromkeys() method.

```
#!/usr/bin/python3
seq = ('name', 'age', 'sex')
dict = dict.fromkeys(seq)
print ("New Dictionary : %s" %  str(dict))
dict = dict.fromkeys(seq, 10)
print ("New Dictionary : %s" %  str(dict))
```

When we run the above program, it produces the following result-

```
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

Dictionary get() Method

Description

The method **get()** returns a value for the given key. If the key is not available then returns default value None.

Syntax

Following is the syntax for get() method-

```
dict.get(key, default=None)
```

Parameters

- **key** - This is the Key to be searched in the dictionary.
- **default** - This is the Value to be returned in case key does not exist.

Return Value

This method returns a value for the given key. If the key is not available, then returns default value as None.

Example

The following example shows the usage of get() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 27}
print ("Value : %s" %  dict.get('Age'))
print ("Value : %s" %  dict.get('Sex', "NA"))
```

When we run the above program, it produces the following result-

```
Value : 27
Value : NA
```

Dictionary items() Method

Description

The method items() returns a list of dict's (key, value) tuple pairs.

Syntax

Following is the syntax for items() method-

```
dict.items()
```

Parameters

NA

Return Value

This method returns a list of tuple pairs.

Example

The following example shows the usage of items() method.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.items())
```

When we run the above program, it produces the following result-

```
Value : [('Age', 7), ('Name', 'Zara')]
```

Dictionary keys() Method

Description

The method **keys()** returns a list of all the available keys in the dictionary.

Syntax

Following is the syntax for keys() method-

```
dict.keys()
```

Parameters

NA

Return Value

This method returns a list of all the available keys in the dictionary.

Example

The following example shows the usage of keys() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.keys())
```

When we run the above program, it produces the following result-

```
Value : ['Age', 'Name']
```

Dictionary setdefault() Method

Description

The method setdefault() is similar to get(), but will set dict[key]=default if the key is not already in dict.

Syntax

Following is the syntax for setdefault() method-

```
dict.setdefault(key, default=None)
```

Parameters

- **key** - This is the key to be searched.
- **default** - This is the Value to be returned in case key is not found.

Return Value

This method returns the key value available in the dictionary and if given key is not available then it will return provided default value.

Example

The following example shows the usage of setdefault() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.setdefault('Age', None))
print ("Value : %s" %  dict.setdefault('Sex', None))
print (dict)
```

When we run the above program, it produces the following result-

```
Value : 7
Value : None
{'Name': 'Zara', 'Sex': None, 'Age': 7}
```

Dictionary update() Method

Description

The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

Syntax

Following is the syntax for update() method-

```
dict.update(dict2)
```

Parameters

dict2 - This is the dictionary to be added into dict.

Return Value

This method does not return any value.

Example

The following example shows the usage of update() method.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7}
dict2 = {'Sex': 'female' }
dict.update(dict2)
print ("updated dict : ", dict)
```

When we run the above program, it produces the following result-

```
updated dict :  {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
```

Dictionary values() Method

Description

The method **values()** returns a list of all the values available in a given dictionary.

Syntax

Following is the syntax for values() method-

```
dict.values()
```

Parameters

NA

Return Value

This method returns a list of all the values available in a given dictionary.

Example

The following example shows the usage of values() method.

```
#!/usr/bin/python3
dict = {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}
```

```
print ("Values : ",  list(dict.values()))
```

When we run above program, it produces following result-

```
Values :  ['female', 7, 'Zara']
```

14. Python 3 – Date & Time

A Python program can handle date and time in several ways. Converting between date formats is a common chore for computers. Python's time and calendar modules help track dates and times.

What is Tick?

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch).

There is a popular **time** module available in Python, which provides functions for working with times, and for converting between representations. The function **time.time()** returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

Example

```
#!/usr/bin/python3
import time;  # This is required to include time module.

ticks = time.time()
print ("Number of ticks since 12:00am, January 1, 1970:", ticks)
```

This would produce a result something as follows-

```
Number of ticks since 12:00am, January 1, 1970: 1455508609.34375
```

Date arithmetic is easy to do with ticks. However, dates before the epoch cannot be represented in this form. Dates in the far future also cannot be represented this way - the cutoff point is sometime in 2038 for UNIX and Windows.

What is TimeTuple?

Many of the Python's time functions handle time as a tuple of 9 numbers, as shown below-

Index	Field	Values
0	4-digit year	2016
1	Month	1 to 12
2	Day	1 to 31

3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 61 (60 or 61 are leap-seconds)
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight savings	-1, 0, 1, -1 means library determines DST

For Example-

```
>>>import time
>>> print (time.localtime())
```

This would produce a result as follows-

```
time.struct_time(tm_year=2016,  tm_mon=2,  tm_mday=15,  tm_hour=9,  tm_min=29,
tm_sec=2, tm_wday=0, tm_yday=46, tm_isdst=0)
```

The above tuple is equivalent to **struct_time** structure. This structure has the following attributes-

Index	Attributes	Values
0	tm_year	2016
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6 (0 is Monday)

7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

Getting current time

To translate a time instant from **seconds** since the epoch floating-point value into a time-tuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all valid nine items.

```
#!/usr/bin/python3
import time

localtime = time.localtime(time.time())
print ("Local current time :", localtime)
```

This would produce the following result, which could be formatted in any other presentable form-

```
Local current time : time.struct_time(tm_year=2016, tm_mon=2, tm_mday=15,
tm_hour=9, tm_min=29, tm_sec=2, tm_wday=0, tm_yday=46, tm_isdst=0)
```

Getting formatted time

You can format any time as per your requirement, but a simple method to get time in a readable format is **asctime()** –

```
#!/usr/bin/python3
import time

localtime = time.asctime( time.localtime(time.time()) )
print ("Local current time :", localtime)
```

This would produce the following result-

```
Local current time : Mon Feb 15 09:34:03 2016
```

Getting calendar for a month

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (Jan 2008).

```
#!/usr/bin/python3
import calendar
```

```
cal = calendar.month(2016, 2)
print ("Here is the calendar:")
print (cal)
```

This would produce the following result-

```
Here is the calendar:
   February 2016
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29
```

The time Module

There is a popular **time** module available in Python, which provides functions for working with times and for converting between representations. Here is the list of all available methods.

SN	Function with Description
1	**time.altzone** The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Use this if the daylight is nonzero.
2	**time.asctime([tupletime])** Accepts a time-tuple and returns a readable 24-character string such as 'Tue Dec 11 18:07:14 2008'.
3	**time.clock( )** Returns the current CPU time as a floating-point number of seconds. To measure computational costs of different approaches, the value of time.clock is more useful than that of time.time().
4	**time.ctime([secs])** Like asctime(localtime(secs)) and without arguments is like asctime( )

5	**time.gmtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. Note : t.tm_isdst is always 0
6	**time.localtime([secs])** Accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).
7	**time.mktime(tupletime)** Accepts an instant expressed as a time-tuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.
8	**time.sleep(secs)** Suspends the calling thread for secs seconds.
9	**time.strftime(fmt[,tupletime])** Accepts an instant expressed as a time-tuple in local time and returns a string representing the instant as specified by string fmt.
10	**time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')** Parses str according to format string fmt and returns the instant in time-tuple format.
11	**time.time( )** Returns the current time instant, a floating-point number of seconds since the epoch.
12	**time.tzset()** Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

Let us go through the functions briefly-

Time altzone() Method

Description

The method **altzone()** is the attribute of the time module. This returns the offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local

DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

Syntax

Following is the syntax for altzone() method-

```
time.altzone
```

Parameters

NA

Return Value

This method returns the offset of the local DST timezone, in seconds west of UTC, if one is defined.

Example

The following example shows the usage of altzone() method.

```
#!/usr/bin/python3
import time
print ("time.altzone : ", time.altzone)
```

When we run the above program, it produces the following result-

```
time.altzone :  -23400
```

Time asctime() Method

Description

The method asctime() converts a tuple or struct_time representing a time as returned by gmtime() or localtime() to a 24-character string of the following form: 'Tue Feb 17 23:21:05 2009'.

Syntax

Following is the syntax for asctime() method-

```
time.asctime([t]))
```

Parameters

t - This is a tuple of 9 elements or struct_time representing a time as returned by gmtime() or localtime() function.

Return Value

This method returns 24-character string of the following form: 'Tue Feb 17 23:21:05 2009'.

Example

The following example shows the usage of asctime() method.

```
#!/usr/bin/python3
import time
t = time.localtime()
print ("asctime : ",time.asctime(t))
```

When we run the above program, it produces the following result-

```
asctime :  Mon Feb 15 09:46:24 2016
```

Time clock() Method

Description

The method **clock()** returns the current processor time as a floating point number expressed in seconds on Unix. The precision depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function QueryPerformanceCounter.

Syntax

Following is the syntax for clock() method-

```
time.clock()
```

Parameters

NA

Return Value

This method returns the current processor time as a floating point number expressed in seconds on Unix and in Windows it returns wall-clock seconds elapsed since the first call to this function, as a floating point number.

Example

The following example shows the usage of clock() method.

```
#!/usr/bin/python3
import time
def procedure():
    time.sleep(2.5)
# measure process time
t0 = time.clock()
procedure()
print (time.clock() - t0, "seconds process time")
# measure wall time
t0 = time.time()
procedure()
print (time.time() - t0, "seconds wall time")
```

When we run the above program, it produces the following result-

```
2.4993855364299096 seconds process time
2.5 seconds wall time
```

Note: Not all systems can measure the true process time. On such systems (including Windows), clock usually measures the wall time since the program was started.

Time ctime() Method

Description

The method ctime() converts a time expressed in seconds since the epoch to a string representing local time. If secs is not provided or None, the current time as returned by time() is used. This function is equivalent to asctime(localtime(secs)). Locale information is not used by ctime().

Syntax

Following is the syntax for ctime() method-

```
time.ctime([ sec ])
```

Parameters

sec - These are the number of seconds to be converted into string representation.

Return Value

This method does not return any value.

Example

The following example shows the usage of ctime() method.

```
#!/usr/bin/python3
import time
print ("ctime : ", time.ctime())
```

When we run the above program, it produces the following result-

```
ctime :  Mon Feb 15 09:55:34 2016
```

Time gmtime() Method

Description

The method gmtime() converts a time expressed in seconds since the epoch to a struct_time in UTC in which the dst flag is always zero. If secs is not provided or None, the current time as returned by time() is used.

Syntax

Following is the syntax for gmtime() method-

```
time.gmtime([ sec ])
```

Parameters

sec - These are the number of seconds to be converted into structure struct_time representation.

Return Value

This method does not return any value.

Example

The following example shows the usage of gmtime() method.

```
#!/usr/bin/python3
import time
print ("gmtime :", time.gmtime(1455508609.34375))
```

When we run the above program, it produces the following result-

```
gmtime   :   time.struct_time(tm_year=2016,   tm_mon=2,   tm_mday=15,   tm_hour=3,
tm_min=56, tm_sec=49, tm_wday=0, tm_yday=46, tm_isdst=0)
```

Time localtime() Method

Description

The method localtime() is similar to gmtime() but it converts number of seconds to local time. If secs is not provided or None, the current time as returned by time() is used. The dst flag is set to 1 when DST applies to the given time.

Syntax

Following is the syntax for localtime() method-

```
time.localtime([ sec ])
```

Parameters

sec - These are the number of seconds to be converted into structure struct_time representation.

Return Value

This method does not return any value.

Example

The following example shows the usage of localtime() method.

```
#!/usr/bin/python3
import time
print ("time.localtime() : %s" , time.localtime())
```

When we run the above program, it produces the following result-

```
time.localtime()   :   time.struct_time(tm_year=2016,   tm_mon=2,   tm_mday=15,
tm_hour=10, tm_min=13, tm_sec=50, tm_wday=0, tm_yday=46, tm_isdst=0)
```

Time mktime() Method

Description

The method mktime() is the inverse function of localtime(). Its argument is the struct_time or full 9-tuple and it returns a floating point number, for compatibility with time().

If the input value cannot be represented as a valid time, either OverflowError or ValueError will be raised.

Syntax

Following is the syntax for mktime() method-

```
time.mktime(t)
```

Parameters

t - This is the struct_time or full 9-tuple.

Return Value

This method returns a floating point number, for compatibility with time().

Example

The following example shows the usage of mktime() method.

```
#!/usr/bin/python3
import time
t = (2016, 2, 15, 10, 13, 38, 1, 48, 0)
d=time.mktime(t)
print ("time.mktime(t) : %f" %  d)
print ("asctime(localtime(secs)): %s" % time.asctime(time.localtime(d)))
```

When we run the above program, it produces the following result-

```
time.mktime(t) : 1455511418.000000
asctime(localtime(secs)): Mon Feb 15 10:13:38 2016
```

Time sleep() Method

Description

The method **sleep()** suspends execution for the given number of seconds. The argument may be a floating point number to indicate a more precise sleep time.

The actual suspension time may be less than that requested because any caught signal will terminate the sleep() following execution of that signal's catching routine.

Syntax

Following is the syntax for sleep() method-

```
time.sleep(t)
```

Parameters

t - This is the number of seconds for which the execution is to be suspended.

Return Value

This method does not return any value.

Example

The following example shows the usage of sleep() method.

```
#!/usr/bin/python3
import time
print ("Start : %s" % time.ctime())
time.sleep( 5 )
print ("End : %s" % time.ctime())
```

When we run the above program, it produces the following result-

```
Start : Mon Feb 15 12:08:42 2016
End : Mon Feb 15 12:08:47 2016
```

Time strftime() Method

Description

The method **strftime()** converts a tuple or struct_time representing a time as returned by gmtime() or localtime() to a string as specified by the format argument.

If t is not provided, the current time as returned by localtime() is used. The format must be a string. An exception ValueError is raised if any field in t is outside of the allowed range.

Syntax

Following is the syntax for strftime() method-

```
time.strftime(format[, t])
```

Parameters

- **t** - This is the time in number of seconds to be formatted.

- **format** - This is the directive which would be used to format given time.

The following directives can be embedded in the format string-

Directive

- %a - abbreviated weekday name
- %A - full weekday name
- %b - abbreviated month name
- %B - full month name
- %c - preferred date and time representation
- %C - century number (the year divided by 100, range 00 to 99)
- %d - day of the month (01 to 31)
- %D - same as %m/%d/%y
- %e - day of the month (1 to 31)
- %g - like %G, but without the century
- %G - 4-digit year corresponding to the ISO week number (see %V).
- %h - same as %b
- %H - hour, using a 24-hour clock (00 to 23)
- %I - hour, using a 12-hour clock (01 to 12)
- %j - day of the year (001 to 366)
- %m - month (01 to 12)
- %M - minute
- %n - newline character
- %p - either am or pm according to the given time value
- %r - time in a.m. and p.m. notation
- %R - time in 24 hour notation
- %S - second
- %t - tab character
- %T - current time, equal to %H:%M:%S
- %u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1
- %U - week number of the current year, starting with the first Sunday as the first day of the first week

- %V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week
- %W - week number of the current year, starting with the first Monday as the first day of the first week
- %w - day of the week as a decimal, Sunday=0
- %x - preferred date representation without the time
- %X - preferred time representation without the date
- %y - year without a century (range 00 to 99)
- %Y - year including the century
- %Z or %z - time zone or name or abbreviation
- %% - a literal % character

Return Value

This method does not return any value.

Example

The following example shows the usage of strftime() method.

```
#!/usr/bin/python3
import time
t = (2015, 12, 31, 10, 39, 45, 1, 48, 0)
t = time.mktime(t)
print (time.strftime("%b %d %Y %H:%M:%S", time.localtime(t)))
```

When we run the above program, it produces the following result-

```
Dec 31 2015 10:39:45
```

Time strptime() Method

Description

The method strptime() parses a string representing a time according to a format. The return value is a struct_time as returned by gmtime() or localtime().

The format parameter uses the same directives as those used by strftime(); it defaults to "%a %b %d %H:%M:%S %Y" which matches the formatting returned by ctime().

If string cannot be parsed according to format, or if it has excess data after parsing, ValueError is raised.

Syntax

Following is the syntax for strptime() method-

```
time.strptime(string[, format])
```

Parameters

- **string** - This is the time in string format which would be parsed based on the given format.
- **format** - This is the directive which would be used to parse the given string.

Directive

The following directives can be embedded in the format string-

- %a - abbreviated weekday name
- %A - full weekday name
- %b - abbreviated month name
- %B - full month name
- %c - preferred date and time representation
- %C - century number (the year divided by 100, range 00 to 99)
- %d - day of the month (01 to 31)
- %D - same as %m/%d/%y
- %e - day of the month (1 to 31)
- %g - like %G, but without the century
- %G - 4-digit year corresponding to the ISO week number (see %V).
- %h - same as %b
- %H - hour, using a 24-hour clock (00 to 23)
- %I - hour, using a 12-hour clock (01 to 12)
- %j - day of the year (001 to 366)
- %m - month (01 to 12)
- %M - minute
- %n - newline character
- %p - either am or pm according to the given time value
- %r - time in a.m. and p.m. notation
- %R - time in 24 hour notation
- %S - second
- %t - tab character
- %T - current time, equal to %H:%M:%S
- %u - weekday as a number (1 to 7), Monday=1. Warning: In Sun Solaris Sunday=1
- %U - week number of the current year, starting with the first Sunday as the first day of the first week

- %V - The ISO 8601 week number of the current year (01 to 53), where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week
- %W - week number of the current year, starting with the first Monday as the first day of the first week
- %w - day of the week as a decimal, Sunday=0
- %x - preferred date representation without the time
- %X - preferred time representation without the date
- %y - year without a century (range 00 to 99)
- %Y - year including the century
- %Z or %z - time zone or name or abbreviation
- %% - a literal % character

Return Value

This return value is struct_time as returned by gmtime() or localtime().

Example

The following example shows the usage of strptime() method.

```
#!/usr/bin/python3
import time
struct_time = time.strptime("30 12 2015", "%d %m %Y")
print ("tuple : ", struct_time)
```

When we run the above program, it produces the following result-

```
tuple : time.struct_time(tm_year=2015, tm_mon=12, tm_mday=30, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=2, tm_yday=364, tm_isdst=-1)
```

Time time() Method

Description

The method time() returns the time as a floating point number expressed in seconds since the epoch, in UTC.

Note: Even though the time is always returned as a floating point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

Syntax

Following is the syntax for time() method-

```
time.time()
```

Parameters

NA

Return Value

This method returns the time as a floating point number expressed in seconds since the epoch, in UTC.

Example

The following example shows the usage of time() method.

```
#!/usr/bin/python3
import time
print ("time.time(): %f " %  time.time())
print (time.localtime( time.time() ))
print (time.asctime( time.localtime(time.time()) ))
```

When we run the above program, it produces the following result-

```
time.time(): 1455519806.011433
time.struct_time(tm_year=2016,  tm_mon=2,  tm_mday=15,  tm_hour=12,  tm_min=33,
tm_sec=26, tm_wday=0, tm_yday=46, tm_isdst=0)
Mon Feb 15 12:33:26 2016
```

Time tzset() Method

Description

The method **tzset()** resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is done.

The standard format of the TZ environment variable is (whitespace added for clarity)-

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

- **std and dst:** Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname.

- **offset:** The offset has the form: .hh[:mm[:ss]]. This indicates the value added the local time to arrive at UTC. If preceded by a '-', the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

- **start[/time], end[/time]:** Indicates when to change to and back from DST. The format of the start and end dates are one of the following:
  - **Jn**: The Julian day n (1 <= n <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.
  - **n**: The zero-based Julian day (0 <= n <= 365). Leap days are counted, and it is possible to refer to February 29.
  - **Mm.n.d**: The d'th day (0 <= d <= 6) or week n of month m of the year (1 <= n <= 5, 1 <= m <= 12, where week 5 means 'the last d day in month m' which may occur in either the fourth or the fifth week). Week 1 is the first week in which the d'th day occurs. Day zero is Sunday.
  - **time**: This has the same format as offset except that no leading sign ('-' or '+') is allowed. The default, if time is not given, is 02:00:00.

Syntax

Following is the syntax for tzset() method-

```
time.tzset()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of tzset() method.

```
#!/usr/bin/python3
import time
import os

os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
time.tzset()
print time.strftime('%X %x %Z')

os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
time.tzset()
print time.strftime('%X %x %Z')
```

When we run the above program, it produces the following result-

```
13:00:40 02/17/09 EST
05:00:40 02/18/09 AEDT
```

There are two important attributes available with time module. They are-

SN	Attribute with Description
1	**time.timezone** Attribute time.timezone is the offset in seconds of the local time zone (without DST) from UTC (>0 in the Americas; <=0 in most of Europe, Asia, Africa).
2	**time.tzname** Attribute time.tzname is a pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

The calendar Module

The calendar module supplies calendar-related functions, including functions to print a text calendar for a given month or year.

By default, calendar takes Monday as the first day of the week and Sunday as the last one. To change this, call the **calendar.setfirstweekday()** function.

Here is a list of functions available with the **calendar** module-

SN	Function with Description
1	**calendar.calendar(year,w=2,l=1,c=6)** Returns a multiline string with a calendar for year year formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21*w+18+2*c. l is the number of lines for each week.

2	**calendar.firstweekday( )** Returns the current setting for the weekday that starts each week. By default, when calendar is first imported, this is 0, meaning Monday.
3	**calendar.isleap(year)** Returns True if year is a leap year; otherwise, False.
4	**calendar.leapdays(y1,y2)** Returns the total number of leap days in the years within range(y1,y2).
5	**calendar.month(year,month,w=2,l=1)** Returns a multiline string with a calendar for month month of year year, one line per week plus two header lines. w is the width in characters of each date; each line has length 7*w+6. l is the number of lines for each week.
6	**calendar.monthcalendar(year,month)** Returns a list of lists of ints. Each sublist denotes a week. Days outside month month of year year are set to 0; days within the month are set to their day-of-month, 1 and up.
7	**calendar.monthrange(year,month)** Returns two integers. The first one is the code of the weekday for the first day of the month month in year year; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 to 12.
8	**calendar.prcal(year,w=2,l=1,c=6)** Like print calendar.calendar(year,w,l,c).
9	**calendar.prmonth(year,month,w=2,l=1)** Like print calendar.month(year,month,w,l).
10	**calendar.setfirstweekday(weekday)** Sets the first day of each week to weekday code weekday. Weekday codes are 0 (Monday) to 6 (Sunday).
11	**calendar.timegm(tupletime)** The inverse of time.gmtime: accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.

12	**calendar.weekday(year,month,day)** Returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

Other Modules & Functions

If you are interested, then here you would find a list of other important modules and functions to play with date & time in Python-

- The datetime Module
- The pytz Module
- The dateutil Module

15. Python 3 – Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on the standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
```

```
    return
```

Calling a Function

Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is an example to call the **printme()** function-

```
#!/usr/bin/python3

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return

# Now you can call printme function
printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result-

```
This is first call to the user defined function!
Again second call to the same function
```

Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example-

```
#!/usr/bin/python3

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   print ("Values inside the function before change: ", mylist)
   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
```

```
   return
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Here, we are maintaining reference of the passed object and appending values in the same object. Therefore, this would produce the following result-

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python3

# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4] # This would assi new reference in mylist
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result-

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments-

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows-

```
#!/usr/bin/python3
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result-

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme()
TypeError: printme() missing 1 required positional argument: 'str'
```

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways-

```
#!/usr/bin/python3


# Function definition is here
```

```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result-

```
My string
```

The following example gives a clearer picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python3

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Age ", age)
   return

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result-

```
Name:  miki
Age  50
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed.

```
#!/usr/bin/python3

# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print ("Name: ", name)
```

```
    print ("Age ", age)
    return
# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result-

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is given below-

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example-

```
#!/usr/bin/python3

# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print ("Output is: ")
    print (arg1)
    for var in vartuple:
        print (var)
    return
# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result-

```
Output is:
10
Output is:
70
60
50
```

The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

Syntax

The syntax of lambda function contains only a single statement, which is as follows-

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Following is an example to show how lambda form of function works-

```
#!/usr/bin/python3


# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2


 # Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

When the above code is executed, it produces the following result-

```
Value of total :  30
Value of total :  40
```

The return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the examples given above are not returning any value. You can return a value from a function as follows-

```
#!/usr/bin/python3

# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print ("Inside the function : ", total)
   return total

# Now you can call sum function
total = sum( 10, 20 )
print ("Outside the function : ", total )
```

When the above code is executed, it produces the following result-

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python-

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example-

```
#!/usr/bin/python3


total = 0 # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print ("Inside the function local total : ", total)
   return total


# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total )
```

When the above code is executed, it produces the following result-

```
Inside the function local total :  30
Outside the function global total :  0
```

16. Python 3 – Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example

The Python code for a module named aname normally resides in a file namedaname.py. Here is an example of a simple module, support.py-

```
def print_func( par ):
   print "Hello : ", par
   return
```

The import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax-

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script-

```
#!/usr/bin/python3
# Import module support
import support
# Now you can call defined function that module as follows
support.print_func("Zara")
```

When the above code is executed, it produces the following result-

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening repeatedly, if multiple imports occur.

The from...import Statement

Python's **from** statement lets you import specific attributes from a module into the current namespace. The **from...import** has the following syntax-

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement-

```
#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
>>> from fib import fib
>>> fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The from...import * Statement:

It is also possible to import all the names from a module into the current namespace by using the following import statement-

```
from modname import *
```

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Executing Modules as Scripts

Within a module, the module's name (as a string) is available as the value of the global variable __name__. The code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__".

Add this code at the end of your module-

```
#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
if __name__ == "__main__":
    f=fib(100)
    print(f)
```

When you run the above code, the following output will be displayed.

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences-

- The current directory.
- If the module is not found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python3/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The PYTHONPATH Variable

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system-

```
set PYTHONPATH=c:\python34\lib;
```

And here is a typical PYTHONPATH from a UNIX system-

```
set PYTHONPATH=/usr/local/lib/python
```

Namespaces and Scoping

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

- A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.
- Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.
- The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable Money in the global namespace. Within the function Money, we assign Money a value, therefore Python assumes Money as a local variable.

However, we accessed the value of the local variable Money before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python3

Money = 2000
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print (Money)
AddMoney()
print (Money)
```

The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example-

```
#!/usr/bin/python3
# Import built-in module math
import math
content = dir(math)
print (content)
```

When the above code is executed, it produces the following result-

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Here, the special string variable __name__ is the module's name, and __file__is the filename from which the module was loaded.

The globals() and locals() Functions

The **globals() and locals()** functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

- If **locals()** is called from within a function, it will return all the names that can be accessed locally from that function.

- If **globals()** is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the **keys()** function.

The reload() Function

When a module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this-

```
reload(module_name)
```

Here, module_name is the name of the module you want to reload and not the string containing the module name. For example, to reload hello module, do the following-

```
reload(hello)
```

Packages in Python

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file Pots.py available in Phone directory. This file has the following line of source code-

```
#!/usr/bin/python3
def Pots():
   print ("I'm Pots Phone")
```

Similarly, we have other two files having different functions with the same name as above. They are –

- *Phone/Isdn.py* file having function Isdn()
- *Phone/G3.py* file having function G3()

Now, create one more file __init__.py in the *Phone* directory-

- Phone/__init__.py

To make all of your functions available when you have imported Phone, you need to put explicit import statements in __init__.py as follows-

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

After you add these lines to __init__.py, you have all of these classes available when you import the Phone package.

```
#!/usr/bin/python3
# Now import your Phone Package.
import Phone
Phone.Pots()
Phone.Isdn()
Phone.G3()
```

When the above code is executed, it produces the following result-

```
I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone
```

In the above example, we have taken example of a single function in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.

17. Python 3 – Files I/O

This chapter covers all the basic I/O functions available in Python 3. For more functions, please refer to the standard Python documentation.

Printing to the Screen

The simplest way to produce output is using the *print* statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows-

```
#!/usr/bin/python3

print ("Python is really a great language,", "isn't it?")
```

This produces the following result on your standard screen-

```
Python is really a great language, isn't it?
```

Reading Keyboard Input

Python 2 has two built-in functions to read data from standard input, which by default comes from the keyboard. These functions are **input()** and **raw_input()**

In Python 3, raw_input() function is deprecated. Moreover, input() functions read data from keyboard as string, irrespective of whether it is enclosed with quotes ('' or "") or not.

The input Function

The **input([prompt])** function is equivalent to raw_input, except that it assumes that the input is a valid Python expression and returns the evaluated result to you.

```
#!/usr/bin/python3


>>> x=input("something:")

something:10

>>> x

'10'

>>> x=input("something:")

something:'10' #entered data treated as string with or without ''

>>> x

"'10'"
```

Opening and Closing Files

Until now, you have been reading and writing to the standard input and output. Now, we will see how to use actual data files.

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.

The open Function

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

```
file object = open(file_name [, access_mode][, buffering])
```

Here are parameter details-

- **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.

- **access_mode:** The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).

- **buffering:** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

Here is a list of the different modes of opening a file-

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.

rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The file Object Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Here is a list of all the attributes related to a file object-

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.

file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Note: softspace attribute is not supported in Python 3.x

Example

```
#!/usr/bin/python3

# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
fo.close()
```

This produces the following result-

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
```

The close() Method

The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
fileObject.close();
```

Example

```
#!/usr/bin/python3

# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
```

```
# Close opened file
fo.close()
```

This produces the following result-

```
Name of the file:  foo.txt
```

Reading and Writing Files

The file object provides a set of access methods to make our lives easier. We would see how to use read() and write() methods to read and write files.

The write() Method

The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.

The write() method does not add a newline character ('\n') to the end of the string-

Syntax

```
fileObject.write(string);
```

Here, passed parameter is the content to be written into the opened file.

Example

```
#!/usr/bin/python3


# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")


# Close opend file
fo.close()
```

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have the following content-

```
Python is a great language.
Yeah its great!!
```

The read() Method

The read() method reads a string from an open file. It is important to note that Python strings can have binary data apart from the text data.

Syntax

```
fileObject.read([count]);
```

Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.

Example

Let us take a file foo.txt, which we created above.

```
#!/usr/bin/python3


# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)
# Close opened file
fo.close()
```

This produces the following result-

```
Read String is :  Python is
```

File Positions

The tell() method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

The *seek(*offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If *from* is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. If it is set to 2 then the end of the file would be taken as the reference position.

Example

Let us take a file foo.txt, which we created above.

```
#!/usr/bin/python3
```

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)

# Check current position
position = fo.tell()
print ("Current file position : ", position)

# Reposition pointer at the beginning once again
position = fo.seek(0, 0)
str = fo.read(10)
print ("Again read String is : ", str)
# Close opened file
fo.close()
```

This produces the following result-

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Renaming and Deleting Files

Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module, you need to import it first and then you can call any related functions.

The rename() Method

The rename() method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_file_name, new_file_name)
```

Example

Following is an example to rename an existing file *test1.txt*-

```
#!/usr/bin/python3
import os
```

```
# Rename a file from test1.txt to test2.txt
os.rename( "test1.txt", "test2.txt" )
```

The remove() Method

You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax

```
os.remove(file_name)
```

Example

Following is an example to delete an existing file test2.txt-

```
#!/usr/bin/python3
import os

# Delete file test2.txt
os.remove("text2.txt")
```

Directories in Python

All files are contained within various directories, and Python has no problem handling these too. The **os** module has several methods that help you create, remove, and change directories.

The mkdir() Method

You can use the mkdir() method of the **os** module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

Following is an example to create a directory test in the current directory-

```
#!/usr/bin/python3
import os

```

```
# Create a directory "test"
os.mkdir("test")
```

The chdir() Method

You can use the chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

Syntax

```
os.chdir("newdir")
```

Example

Following is an example to go into "/home/newdir" directory-

```
#!/usr/bin/python3
import os

# Changing a directory to "/home/newdir"
os.chdir("/home/newdir")
```

The getcwd() Method

The getcwd() method displays the current working directory.

Syntax

```
os.getcwd()
```

Example

Following is an example to give current directory-

```
#!/usr/bin/python3
import os

# This would give location of the current directory
os.getcwd()
```

The rmdir() Method

The rmdir() method deletes the directory, which is passed as an argument in the method.

Before removing a directory, all the contents in it should be removed.

Syntax

```
os.rmdir('dirname')
```

Example

Following is an example to remove the "/tmp/test" directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

```
#!/usr/bin/python3
import os

# This would  remove "/tmp/test"  directory.
os.rmdir( "/tmp/test"  )
```

File & Directory Related Methods

There are three important sources, which provide a wide range of utility methods to handle and manipulate files & directories on Windows and Unix operating systems. They are as follows-

- **File Object Methods**: The file object provides functions to manipulate files.
- **OS Object Methods**: This provides methods to process files as well as directories.

File Methods

A **file** object is created using *open* function and here is a list of functions which can be called on this object.

S. No.	Methods with Description
1	**file.close()** Close the file. A closed file cannot be read or written any more.
2	**file.flush()**

	Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.
3	**file.fileno()** Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.
4	**file.isatty()** Returns True if the file is connected to a tty(-like) device, else False.
5	**next(file)** Returns the next line from the file each time it is being called.
6	**file.read([size])** Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).
7	**file.readline([size])** Reads one entire line from the file. A trailing newline character is kept in the string.
8	**file.readlines([sizehint])** Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.
9	**file.seek(offset[, whence])** Sets the file's current position
10	**file.tell()** Returns the file's current position
11	**file.truncate([size])** Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
12	**file.write(str)**

	Writes a string to the file. There is no return value.
13	**file.writelines(sequence)** Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

Let us go through the above mentions methods briefly.

File close() Method

Description

The method **close()** closes the opened file. A closed file cannot be read or written any more. Any operation, which requires that the file be opened will raise a ValueError after the file has been closed. Calling close() more than once is allowed.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

Following is the syntax for close() method-

```
fileObject.close()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of close() method.

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
```

File flush() Method

Description

The method **flush()** flushes the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.

Python automatically flushes the files when closing them. But you may want to flush the data before closing any file.

Syntax

Following is the syntax for flush() method-

```
fileObject.flush()
```

Parameters

NA

Return Value

This method does not return any value.

Example

The following example shows the usage of flush() method.

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
# Here it does nothing, but you can call it with read operation.
fo.flush()
# Close opend file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
```

File fileno() Method

Description

The method **fileno()** returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.

Syntax

Following is the syntax for fileno() method-

```
fileObject.fileno()
```

Parameters

NA

Return Value

This method returns the integer file descriptor.

Example

The following example shows the usage of fileno() method.

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
fid = fo.fileno()
print ("File Descriptor: ", fid)
# Close opend file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
File Descriptor:  3
```

File isatty() Method

Description

The method isatty() returns True if the file is connected (is associated with a terminal device) to a tty(-like) device, else False.

Syntax

Following is the syntax for isatty() method-

```
fileObject.isatty()
```

Parameters

NA

Return Value

This method returns true if the file is connected (is associated with a terminal device) to a tty(-like) device, else false.

Example

The following example shows the usage of isatty() method-

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)
ret = fo.isatty()
print ("Return value : ", ret)
# Close opend file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Return value :  False
```

File next() Method

Description

File object in Python 3 does not support **next()** method. Python 3 has a built-in function next() which retrieves the next item from the iterator by calling its __next__() method. If default is given, it is returned if the iterator is exhausted, otherwise StopIteration is raised. This method can be used to read the next input line, from the file object.

Syntax

Following is the syntax for next() method-

```
next(iterator[,default])
```

Parameters

- **iterator** : file object from which lines are to be read
- **default** : returned if iterator exhausted. If not given, StopIteration is raised

Return Value

This method returns the next input line.

Example

The following example shows the usage of next() method-

```
Assuming that 'foo.txt' contains following lines
C++
Java
Python
Perl
PHP
```

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "r")
print ("Name of the file: ", fo.name)
for index in range(5):
   line = next(fo)
   print ("Line No %d - %s" % (index, line))
# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Line No 0 - C++
Line No 1 - Java
Line No 2 - Python
Line No 3 - Perl
Line No 4 - PHP
```

File read() Method

Description

The method **read()** reads at most size bytes from the file. If the read hits EOF before obtaining size bytes, then it reads only available bytes.

Syntax

Following is the syntax for read() method-

```
fileObject.read( size );
```

Parameters

size - This is the number of bytes to be read from the file.

Return Value

This method returns the bytes read in string.

Example

The following example shows the usage of read() method.

```
Assuming that 'foo.txt' file contains the following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)
line = fo.read(10)
print ("Read Line: %s" % (line))
# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
```

```
Read Line: This is 1s
```

File readline() Method

Description

The method readline()reads one entire line from the file. A trailing newline character is kept in the string. If the size argument is present and non-negative, it is a maximum byte count including the trailing newline and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately.

Syntax

Following is the syntax for readline() method-

```
fileObject.readline( size );
```

Parameters

size - This is the number of bytes to be read from the file.

Return Value

This method returns the line read from the file.

Example

The following example shows the usage of readline() method.

Assuming that 'foo.txt' file contains following text-

```
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)
line = fo.readline()
print ("Read Line: %s" % (line))
line = fo.readline(5)
```

```
print ("Read Line: %s" % (line))
# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Read Line: This is 1st line
Read Line: This
```

File readlines() Method

Description

The method **readlines()** reads until EOF using readline() and returns a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

An empty string is returned only when EOF is encountered immediately.

Syntax

Following is the syntax for readlines() method-

```
fileObject.readlines( sizehint );
```

Parameters

sizehint - This is the number of bytes to be read from the file.

Return Value

This method returns a list containing the lines.

Example

The following example shows the usage of readlines() method.

```
Assuming that 'foo.txt' file contains following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)
line = fo.readlines()
print ("Read Line: %s" % (line))
line = fo.readlines(2)
print ("Read Line: %s" % (line))
# Close opened file
fo.close()
```

When we run above program, it produces following result-

```
Name of the file:  foo.txt
Read Line: ['This is 1st line\n', 'This is 2nd line\n',
            'This is 3rd line\n', 'This is 4th line\n',
            'This is 5th line\n']
Read Line:
```

File seek() Method

Description

The method seek() sets the file's current position at the offset. The whence argument is optional and defaults to 0, which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

There is no return value. Note that if the file is opened for appending using either 'a' or 'a+', any seek() operations will be undone at the next write.

If the file is only opened for writing in append mode using 'a', this method is essentially a no-op, but it remains useful for files opened in append mode with reading enabled (mode 'a+').

If the file is opened in text mode using 't', only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

Note that not all file objects are seekable.

Syntax

Following is the syntax for seek() method-

```
fileObject.seek(offset[, whence])
```

Parameters

- **offset**- This is the position of the read/write pointer within the file.
- **whence**- This is optional and defaults to 0 which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

Return Value

This method does not return any value.

Example

The following example shows the usage of seek() method.

```
Assuming that 'foo.txt' file contains following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
# Open a file
fo = open("foo.txt", "rw+")
print ("Name of the file: ", fo.name)

line = fo.readlines()
print ("Read Line: %s" % (line))

# Again set the pointer to the beginning
fo.seek(0, 0)
line = fo.readline()
print ("Read Line: %s" % (line))

# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Read Line: ['This is 1st line\n', 'This is 2nd line\n', 'This is 3rd line\n',
'This is 4th line\n', 'This is 5th line']
Read Line: This is 1st line
```

File tell() Method

Description

The method **tell()** returns the current position of the file read/write pointer within the file.

Syntax

Following is the syntax for tell() method-

```
fileObject.tell()
```

Parameters

NA

Return Value

This method returns the current position of the file read/write pointer within the file.

Example

The following example shows the usage of tell() method-

```
Assuming that 'foo.txt' file contains following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)
line = fo.readline()
print ("Read Line: %s" % (line))
pos=fo.tell()
print ("current position : ",pos)
```

```
# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Read Line: This is 1st line

Current Position: 18
```

File truncate() Method

Description

The method **truncate()** truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.

The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file's current size, the result is platform-dependent.

Note: This method will not work in case the file is opened in read-only mode.

Syntax

Following is the syntax for truncate() method-

```
fileObject.truncate( [ size ])
```

Parameters

size - If this optional argument is present, the file is truncated to (at most) that size.

Return Value

This method does not return any value.

Example

The following example shows the usage of truncate() method.

```
Assuming that 'foo.txt' file contains following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3

fo = open("foo.txt", "r+")
print ("Name of the file: ", fo.name)

line = fo.readline()
print ("Read Line: %s" % (line))

fo.truncate()
line = fo.readlines()
print ("Read Line: %s" % (line))

# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Read Line: This is 1s
Read Line: []
```

File write() Method

Description

The method **write()** writes a string str to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.

Syntax

Following is the syntax for write() method-

```
fileObject.write( str )
```

Parameters

str - This is the String to be written in the file.

Return Value

This method does not return any value.

Example

The following example shows the usage of write() method.

```
Assuming that 'foo.txt' file contains following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
# Open a file in read/write mode
fo = open("abc.txt", "r+")
print ("Name of the file: ", fo.name)
str = "This is 6th line"
# Write a line at the end of the file.
fo.seek(0, 2)
line = fo.write( str )
# Now read complete file from beginning.
fo.seek(0,0)
for index in range(6):
   line = next(fo)
   print ("Line No %d - %s" % (index, line))

# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Line No 0 - This is 1st line
Line No 1 - This is 2nd line
Line No 2 - This is 3rd line
Line No 3 - This is 4th line
Line No 4 - This is 5th line
```

```
Line No 5 - This is 6th line
```

File writelines() Method

Description

The method **writelines()** writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value.

Syntax

Following is the syntax for writelines() method –

```
fileObject.writelines( sequence )
```

Parameters

sequence - This is the Sequence of the strings.

Return Value

This method does not return any value.

Example

The following example shows the usage of writelines() method.

```
Assuming that 'foo.txt' file contains following text:
This is 1st line
This is 2nd line
This is 3rd line
This is 4th line
This is 5th line
```

```
#!/usr/bin/python3
# Open a file in read/write mode
fo = open("abc.txt", "r+")
print ("Name of the file: ", fo.name)
seq = ["This is 6th line\n", "This is 7th line"]
# Write sequence of lines at the end of the file.
fo.seek(0, 2)
line = fo.writelines( seq )
```

```
# Now read complete file from beginning.
fo.seek(0,0)
for index in range(7):
   line = next(fo)
   print ("Line No %d - %s" % (index, line))

# Close opened file
fo.close()
```

When we run the above program, it produces the following result-

```
Name of the file:  foo.txt
Line No 0 - This is 1st line

Line No 1 - This is 2nd line

Line No 2 - This is 3rd line

Line No 3 - This is 4th line

Line No 4 - This is 5th line

Line No 5 - This is 6th line

Line No 6 - This is 7th line
```

OS File/Directory Methods

The **os** module provides a big range of useful methods to manipulate files and directories. Most of the useful methods are listed here:

S. No.	Methods with Description
1	**os.access(path, mode)** Use the real uid/gid to test for access to path.

2	**os.chdir(path)** Change the current working directory to path	
3	**os.chflags(path, flags)** Set the flags of path to the numeric flags.	
4	**os.chmod(path, mode)** Change the mode of path to the numeric mode.	
5	**os.chown(path, uid, gid)** Change the owner and group id of path to the numeric uid and gid.	
6	**os.chroot(path)** Change the root directory of the current process to path.	
7	**os.close(fd)** Close file descriptor fd.	
8	**os.closerange(fd_low, fd_high)** Close all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors.	
9	**os.dup(fd)** Return a duplicate of file descriptor fd.	
10	**os.dup2(fd, fd2)** Duplicate file descriptor fd to fd2, closing the latter first if necessary.	
11	**os.fchdir(fd)**	

	Change the current working directory to the directory represented by the file descriptor fd.
12	**os.fchmod(fd, mode)** Change the mode of the file given by fd to the numeric mode.
13	**os.fchown(fd, uid, gid)** Change the owner and group id of the file given by fd to the numeric uid and gid.
14	**os.fdatasync(fd)** Force write of file with filedescriptor fd to disk.
15	**os.fdopen(fd[, mode[, bufsize]])** Return an open file object connected to the file descriptor fd.
16	**os.fpathconf(fd, name)** Return system configuration information relevant to an open file. name specifies the configuration value to retrieve.
17	**os.fstat(fd)** Return status for file descriptor fd, like stat().
18	**os.fstatvfs(fd)** Return information about the filesystem containing the file associated with file descriptor fd, like statvfs().
19	**os.fsync(fd)** Force write of file with filedescriptor fd to disk.
20	**os.ftruncate(fd, length)**

	Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.
21	**os.getcwd()** Return a string representing the current working directory.
22	**os.getcwdu()** Return a Unicode object representing the current working directory.
23	**os.isatty(fd)** Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.
24	**os.lchflags(path, flags)** Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links.
25	**os.lchmod(path, mode)** Change the mode of path to the numeric mode.
26	**os.lchown(path, uid, gid)** Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links.
27	**os.link(src, dst)** Create a hard link pointing to src named dst.
28	**os.listdir(path)** Return a list containing the names of the entries in the directory given by path.
29	**os.lseek(fd, pos, how)**

	Set the current position of file descriptor fd to position pos, modified by how.
30	**os.lstat(path)** Like stat(), but do not follow symbolic links.
31	**os.major(device)** Extract the device major number from a raw device number.
32	**os.makedev(major, minor)** Compose a raw device number from the major and minor device numbers.
33	**os.makedirs(path[, mode])** Recursive directory creation function.
34	**os.minor(device)** Extract the device minor number from a raw device number .
35	**os.mkdir(path[, mode])** Create a directory named path with numeric mode mode.
36	**os.mkfifo(path[, mode])** Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal).
37	**os.mknod(filename[, mode=0600, device])** Create a filesystem node (file, device special file or named pipe) named filename.
38	**os.open(file, flags[, mode])**

	Open the file file and set various flags according to flags and possibly its mode according to mode.
39	**os.openpty()** Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively.
40	**os.pathconf(path, name)** Return system configuration information relevant to a named file.
41	**os.pipe()** Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.
42	**os.popen(command[, mode[, bufsize]])** Open a pipe to or from command.
43	**os.read(fd, n)** Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.
44	**os.readlink(path)** Return a string representing the path to which the symbolic link points.
45	**os.remove(path)** Remove the file path.
46	**os.removedirs(path)** Remove directories recursively.

47	**os.rename(src, dst)** Rename the file or directory src to dst.
48	**os.renames(old, new)** Recursive directory or file renaming function.
49	**os.rmdir(path)** Remove the directory path
50	**os.stat(path)** Perform a stat system call on the given path.
51	**os.stat_float_times([newvalue])** Determine whether stat_result represents time stamps as float objects.
52	**os.statvfs(path)** Perform a statvfs system call on the given path.
53	**os.symlink(src, dst)** Create a symbolic link pointing to src named dst.
54	**os.tcgetpgrp(fd)** Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()).
55	**os.tcsetpgrp(fd, pg)** Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg.
56	**os.tempnam([dir[, prefix]])**

	Return a unique path name that is reasonable for creating a temporary file.
57	**os.tmpfile()** Return a new file object opened in update mode (w+b).
58	**os.tmpnam()** Return a unique path name that is reasonable for creating a temporary file.
59	**os.ttyname(fd)** Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.
60	**os.unlink(path)** Remove the file path.
61	**os.utime(path, times)** Set the access and modified times of the file specified by path.
62	**os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])** Generate the file names in a directory tree by walking the tree either top-down or bottom-up.
63	**os.write(fd, str)** Write the string str to file descriptor fd. Return the number of bytes actually written.

Let us go through the methods briefly-

os.access() Method

Description

The method access() uses the real uid/gid to test for access to path. Most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to path.It returns True if access is allowed, False if not.

Syntax

Following is the syntax for access() method-

```
os.access(path, mode)
```

Parameters

- **path** - This is the path which would be tested for existence or any access.

- **mode** - This should be F_OK to test the existence of path, or it can be the inclusive OR of one or more of R_OK, W_OK, and X_OK to test permissions.

  - **os.F_OK:** Value to pass as the mode parameter of access() to test the existence of path.
  - **os.R_OK:** Value to include in the mode parameter of access() to test the readability of path.
  - **os.W_OK:** Value to include in the mode parameter of access() to test the writability of path.
  - **os.X_OK:** Value to include in the mode parameter of access() to determine if path can be executed.

Return Value

This method returns True if access is allowed, False if not.

Example

The following example shows the usage of access() method.

```
#!/usr/bin/python3
import os, sys
# Assuming /tmp/foo.txt exists and has read/write permissions.
ret = os.access("/tmp/foo.txt", os.F_OK)
print ("F_OK - return value %s"% ret)

ret = os.access("/tmp/foo.txt", os.R_OK)
print ("R_OK - return value %s"% ret)
```

```
ret = os.access("/tmp/foo.txt", os.W_OK)
print ("W_OK - return value %s"% ret)

ret = os.access("/tmp/foo.txt", os.X_OK)
print ("X_OK - return value %s"% ret)
```

When we run the above program, it produces the following result-

```
F_OK - return value True
R_OK - return value True
W_OK - return value True
X_OK - return value False
```

os.chdir() Method

Description

The method **chdir()** changes the current working directory to the given path.It returns None in all the cases.

Syntax

Following is the syntax for chdir() method-

```
os.chdir(path)
```

Parameters

path - This is complete path of the directory to be changed to a new location.

Return Value

This method does not return any value. It throws FileNotFoundError if the specified path is not found.

Example

The following example shows the usage of chdir() method.

```
#!/usr/bin/python3
import os
path = "d:\\python3" #change path for linux
# Now change the directory
os.chdir( path )
# Check current working directory.
```

```
retval = os.getcwd()
print ("Directory changed successfully %s" % retval)
```

When we run the above program, it produces the following result-

```
Directory changed successfully d:\python3
```

os.chflags() Method

Description

The method **chflags()** sets the flags of path to the numeric flags. The flags may take a combination (bitwise OR) of the various values described below.

Note: This method is available Python version 2.6 onwards. Most of the flags can be changed by super-user only.

Syntax

Following is the syntax for chflags() method-

```
os.chflags(path, flags)
```

Parameters

- **path** - This is a complete path of the directory to be changed to a new location.
- **flags** - The flags specified are formed by OR'ing the following values-
    - os.UF_NODUMP: Do not dump the file.
    - os.UF_IMMUTABLE: The file may not be changed.
    - os.UF_APPEND: The file may only be appended to.
    - os.UF_NOUNLINK: The file may not be renamed or deleted.
    - os.UF_OPAQUE: The directory is opaque when viewed through a union stack.
    - os.SF_ARCHIVED: The file may be archived.
    - os.SF_IMMUTABLE: The file may not be changed.
    - os.SF_APPEND: The file may only be appended to.
    - os.SF_NOUNLINK: The file may not be renamed or deleted.
    - os.SF_SNAPSHOT: The file is a snapshot file.

Return Value

This method does not return any value.

Example

The following example shows the usage of chflags() method.

```
#!/usr/bin/python3
import os
path = "/tmp/foo.txt"
# Set a flag so that file may not be renamed or deleted.
flags = os.SF_NOUNLINK
retval = os.chflags( path, flags)
print ("Return Value: %s" % retval)
```

When we run the above program, it produces the following result-

```
Return Value : None
```

os.chmod() Method

Description

The method chmod() changes the mode of path to the passed numeric mode. The mode may take one of the following values or bitwise ORed combinations of them-

- stat.S_ISUID: Set user ID on execution.
- stat.S_ISGID: Set group ID on execution.
- stat.S_ENFMT: Record locking enforced.
- stat.S_ISVTX: Save text image after execution.
- stat.S_IREAD: Read by owner.
- stat.S_IWRITE: Write by owner.
- stat.S_IEXEC: Execute by owner.
- stat.S_IRWXU: Read, write, and execute by owner.
- stat.S_IRUSR: Read by owner.
- stat.S_IWUSR: Write by owner.
- stat.S_IXUSR: Execute by owner.
- stat.S_IRWXG: Read, write, and execute by group.
- stat.S_IRGRP: Read by group.
- stat.S_IWGRP: Write by group.

- stat.S_IXGRP: Execute by group.
- stat.S_IRWXO: Read, write, and execute by others.
- stat.S_IROTH: Read by others.
- stat.S_IWOTH: Write by others.
- stat.S_IXOTH: Execute by others.

Syntax

Following is the syntax for chmod() method-

```
os.chmod(path, mode)
```

Parameters

- **path** - This is the path for which mode would be set.
- **mode** - This may take one of the above mentioned values or bitwise ORed combinations of them.

Return Value

This method does not return any value.

Note : Although Windows supports chmod(), you can only set the file's read-only flag with it (via the stat.S_IWRITE and stat.S_IREAD constants or a corresponding integer value). All other bits are ignored.

Example

The following example shows the usage of chmod() method-

```
#!/usr/bin/python3
import os, sys, stat

# Assuming /tmp/foo.txt exists, Set a file execute by the group.

os.chmod("/tmp/foo.txt", stat.S_IXGRP)

# Set a file write by others.
os.chmod("/tmp/foo.txt", stat.S_IWOTH)

print ("Changed mode successfully!!")
```

When we run the above program, it produces the following result-

```
Changed mode successfully!!
```

os.chown() Method

Description

The method **chown()** changes the owner and group id of path to the numeric uid and gid. To leave one of the ids unchanged, set it to -1.To set ownership, you would need super user privilege..

Syntax

Following is the syntax for chown() method-

```
os.chown(path, uid, gid)
```

Parameters

- **path** - This is the path for which owner id and group id need to be setup.
- **uid** - This is Owner ID to be set for the file.
- **gid** - This is Group ID to be set for the file.

Return Value

This method does not return any value.

Example

The following example shows the usage of chown() method.

```
#!/usr/bin/python3
import os, sys
# Assuming /tmp/foo.txt exists.
# To set owner ID 100 following has to be done.
os.chown("/tmp/foo.txt", 100, -1)
print ("Changed ownership successfully!!")
```

When we run the above program, it produces the following result-

```
Changed ownership successfully!!
```

os.chroot() Method

Description

The method **chroot()** changes the root directory of the current process to the given path. Available on Unix like systems only. To use this method, you would need super user privilege.

Syntax

Following is the syntax for chroot() method-

```
os.chroot(path)
```

Parameters

path - This is the path which would be set as root for the current process.

Return Value

This method does not return any value.

Example

The following example shows the usage of **chroot()** method.

```
#!/usr/bin/python3
import os, sys
# To set the current root path to /tmp/user
os.chroot("/tmp/usr")
print ("Changed root path successfully!!")
```

When we run the above program, it produces the following result-

```
Changed root path successfully!!
```

Python os.close() Method

Description

The method close() closes the associated with file descriptor fd.

Syntax

Following is the syntax for close() method-

```
os.close(fd)
```

Parameters

fd - This is the file descriptor of the file.

Return Value

This method does not return any value.

Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by os.open() or pipe().

Example

The following example shows the usage of close() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Write one string
line="this is test"
# string needs to be converted byte object
b=str.encode(line)
os.write(fd, b)
# Close opened file
os.close( fd )
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Closed the file successfully!!
```

os.closerange() Method

Description

The method closerange() closes all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors.This method is introduced in Python version 2.6.

Syntax

Following is the syntax for closerange() method-

```
os.closerange(fd_low, fd_high)
```

Parameters

- **fd_low** - This is the Lowest file descriptor to be closed.
- **fd_high** - This is the Highest file descriptor to be closed.

This function is equivalent to-

```
for fd in xrange(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

Return Value

This method does not return any value.

Example

The following example shows the usage of closerange() method.

```
#!/usr/bin/python3
import os, sys

# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )

# Write one string
line="this is test"
# string needs to be converted byte object
b=str.encode(line)
os.write(fd, b)
# Close a single opened file
os.closerange( fd, fd)

print ("Closed all the files successfully!!")
```

This would create given file foo.txt and then write given content in that file. This will produce the following result-

```
Closed all the files successfully!
```

os.dup() Method

Description

The method dup() returns a duplicate of file descriptor **fd** which can be used in place of original descriptor.

Syntax

Following is the syntax for dup() method-

```
os.dup(fd)
```

Parameters

fd - This is the original file descriptor.

Return Value

This method returns a duplicate of file descriptor.

Example

The following example shows the usage of dup() method-

```
#!/usr/bin/python3
import os, sys

# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )

# Get one duplicate file descriptor
d_fd = os.dup( fd )

# Write one string using duplicate fd

line="this is test"
# string needs to be converted byte object
b=str.encode(line)

os.write(d_fd, b)

# Close a single opened file
os.closerange( fd, d_fd)
```

```
print "Closed all the files successfully!!"
```

When we run the above program, it produces the following result-

```
Closed all the files successfully!!
```

os.dup2() Method

Description

The method dup2() duplicates file descriptor fd to fd2, closing the latter first if necessary.

Note: New file description would be assigned only when it is available. In the following example given below, 1000 would be assigned as a duplicate fd in case when 1000 is available.

Syntax

Following is the syntax for dup2() method-

```
os.dup2(fd, fd2)
```

Parameters

- **fd** - This is File descriptor to be duplicated.
- **fd2** - This is Duplicate file descriptor.

Return Value

This method returns a duplicate of file descriptor.

Example

The following example shows the usage of dup2() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )

# Write one string using duplicate fd

line="this is test"
# string needs to be converted byte object
b=str.encode(line)
```

```
os.write(fd, b)

# Now duplicate this file descriptor as 1000
fd2 = 1000
os.dup2(fd, fd2);

# Now read this file from the beginning using fd2.
os.lseek(fd2, 0, 0)
line = os.read(fd2, 100)
str=line.decode()
print ("Read String is : ", str)

# Close opened file
os.closerange( fd,fd2 )

print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Read String is :  This is test
Closed the file successfully!!
```

os.fchdir() Method

Description

The method **fchdir()** change the current working directory to the directory represented by the file descriptor fd. The descriptor must refer to an opened directory, not an open file.

Syntax

Following is the syntax for fchdir() method-

```
os.fchdir(fd)
```

Parameters

fd - This is Directory descriptor.

Return Value

This method does not return any value.

Example

The following example shows the usage of fchdir() method.

```
#!/usr/bin/python3
import os, sys
# First go to the "/var/www/html" directory
os.chdir("/var/www/html" )

# Print current working directory
print ("Current working dir : %s" % os.getcwd())

# Now open a directory "/tmp"
fd = os.open( "/tmp", os.O_RDONLY )

# Use os.fchdir() method to change the dir
os.fchdir(fd)

# Print current working directory
print ("Current working dir : %s" % os.getcwd())

# Close opened directory.
os.close( fd )
```

When we run the above program, it produces the following result-

```
Current working dir : /var/www/html
Current working dir : /tmp
```

os.fchmod() Method

Description

The method fchmod() changes the mode of the file given by fd to the numeric mode. The mode may take one of the following values or bitwise ORed combinations of them-

Note: This method is available Python 2.6 onwards.

- stat.S_ISUID: Set user ID on execution.
- stat.S_ISGID: Set group ID on execution.
- stat.S_ENFMT: Record locking enforced.

- stat.S_ISVTX: Save text image after execution.
- stat.S_IREAD: Read by owner.
- stat.S_IWRITE: Write by owner.
- stat.S_IEXEC: Execute by owner.
- stat.S_IRWXU: Read, write, and execute by owner.
- stat.S_IRUSR: Read by owner.
- stat.S_IWUSR: Write by owner.
- stat.S_IXUSR: Execute by owner.
- stat.S_IRWXG: Read, write, and execute by group.
- stat.S_IRGRP: Read by group.
- stat.S_IWGRP: Write by group.
- stat.S_IXGRP: Execute by group.
- stat.S_IRWXO: Read, write, and execute by others.
- stat.S_IROTH: Read by others.
- stat.S_IWOTH: Write by others.
- stat.S_IXOTH: Execute by others.

Syntax

Following is the syntax for fchmod() method-

```
os.fchmod(fd, mode)
```

Parameters

- **fd** - This is the file descriptor for which mode would be set.
- **mode** - This may take one of the above mentioned values or bitwise ORed combinations of them.

Return Value

This method does not return any value. Available on Unix like operating systems only.

Example

The following example shows the usage of fchmod() method-

```
#!/usr/bin/python3
import os, sys, stat
# Now open a file "/tmp/foo.txt"
fd = os.open( "/tmp", os.O_RDONLY )

# Set a file execute by the group.

os.fchmod( fd, stat.S_IXGRP)

# Set a file write by others.
os.fchmod(fd, stat.S_IWOTH)

print ("Changed mode successfully!!")

# Close opened file.
os.close( fd )
```

When we run the above program, it produces the following result-

```
Changed mode successfully!!
```

os.fchown() Method

Description

The method fchown() changes the owner and group id of the file given by fd to the numeric uid and gid. To leave one of the ids unchanged, set it to -1.

Note: This method is available Python 2.6 onwards.

Syntax

Following is the syntax for fchown() method-

```
os.fchown(fd, uid, gid)
```

Parameters

- **fd** - This is the file descriptor for which owner id and group id need to be set up.
- **uid** - This is Owner ID to be set for the file.

- **gid** - This is Group ID to be set for the file.

Return Value

This method does not return any value. Available in Unix like operating systems only.

Example

The following example shows the usage of fchown() method.

```
#!/usr/bin/python3
import os, sys, stat
# Now open a file "/tmp/foo.txt"
fd = os.open( "/tmp", os.O_RDONLY )
# Set the user Id to 100 for this file.
os.fchown( fd, 100, -1)

# Set the group Id to 50 for this file.
os.fchown( fd, -1, 50)

print ("Changed ownership successfully!!")

# Close opened file.
os.close( fd )
```

When we run the above program, it produces the following result-

```
Changed ownership successfully!!
```

os.fdatasync() Method

Description

The method fdatasync() forces write of file with filedescriptor fd to disk. This does not force update of metadata. If you want to flush your buffer then you can use this method.

Syntax

Following is the syntax for fdatasync() method-

```
os.fdatasync(fd)
```

Parameters

fd - This is the file descriptor for which data to be written.

Return Value

This method does not return any value.

Example

The following example shows the usage of fdatasync() method-

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Write one string
line="this is test"
# string needs to be converted byte object
b=str.encode(line)
os.write(fd, b)
# Now you can use fdatasync() method.
# Infact here you would not be able to see its effect.
os.fdatasync(fd)

# Now read this file from the beginning.
os.lseek(fd, 0, 0)
str = os.read(fd, 100)
line = os.read(fd2, 100)
str=line.decode()
print ("Read String is : ", str)

# Close opened file
os.close( fd )

print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Read String is :  This is test
Closed the file successfully!!
```

os.fdopen() Method

Description

The method **fdopen()** returns an open file object connected to the file descriptor fd. Then you can perform all the defined functions on file object.

Syntax

Following is the syntax for fdopen() method-

```
os.fdopen(fd, [, mode[, bufsize]]);
```

Parameters

- **fd** - This is the file descriptor for which a file object is to be returned.
- **mode** - This optional argument is a string indicating how the file is to be opened. The most commonly-used values of mode are 'r' for reading, 'w' for writing (truncating the file if it already exists), and 'a' for appending.
- **bufsize** - This optional argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size.

Return Value

This method returns an open file object connected to the file descriptor.

Example

The following example shows the usage of fdopen() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )

# Now get a file object for the above file.
fo = os.fdopen(fd, "w+")

# Tell the current position
print ("Current I/O pointer position :%d" % fo.tell())

# Write one string
fo.write( "Python is a great language.\nYeah its great!!\n");
```

```
# Now read this file from the beginning.
os.lseek(fd, 0, 0)
str = os.read(fd, 100)
print ("Read String is : ", str)

# Tell the current position
print "Current I/O pointer position :%d" % fo.tell()

# Close opened file
fo.close()
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Current I/O pointer position :0
Read String is :  This is testPython is a great language.
Yeah its great!!

Current I/O pointer position :45
Closed the file successfully!!
```

os.fpathconf() Method

Description

The method **fpathconf()** returns system configuration information relevant to an open file.This variable is very similar to unix system call fpathconf() and accept the similar arguments.

Syntax

Following is the syntax for fpathconf() method-

```
os.fpathconf(fd, name)
```

Parameters

- **fd** - This is the file descriptor for which system configuration information is to be returned.
- **name** - This specifies the configuration value to retrieve; it may be a string, which is the name of a defined system value; these names are specified in a number of

standards (POSIX.1, Unix 95, Unix 98, and others). The names known to the host operating system are given in the os.pathconf_names dictionary.

Return Value

This method returns system configuration information relevant to an open file.

Example

The following example shows the usage of fpathconf() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
print ("%s" % os.pathconf_names)
# Now get maximum number of links to the file.
no = os.fpathconf(fd, 'PC_LINK_MAX')
print ("Maximum number of links to the file. :%d" % no)
# Now get maximum length of a filename
no = os.fpathconf(fd, 'PC_NAME_MAX')
print ("Maximum length of a filename :%d" % no)

# Close opened file
os.close( fd)
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
{'PC_MAX_INPUT': 2, 'PC_VDISABLE': 8, 'PC_SYNC_IO': 9,
'PC_SOCK_MAXBUF': 12, 'PC_NAME_MAX': 3, 'PC_MAX_CANON': 1,
'PC_PRIO_IO': 11, 'PC_CHOWN_RESTRICTED': 6, 'PC_ASYNC_IO': 10,
'PC_NO_TRUNC': 7, 'PC_FILESIZEBITS': 13, 'PC_LINK_MAX': 0,
'PC_PIPE_BUF': 5, 'PC_PATH_MAX': 4}

Maximum number of links to the file. :127
Maximum length of a filename :255
Closed the file successfully!!
```

os.fstat() Method

Description

The method fstat() returns information about a file associated with the fd. Here is the structure returned by fstat method-

- st_dev: ID of device containing file
- st_ino: inode number
- st_mode: protection
- st_nlink: number of hard links
- st_uid: user ID of owner
- st_gid: group ID of owner
- st_rdev: device ID (if special file)
- st_size: total size, in bytes
- st_blksize: blocksize for filesystem I/O
- st_blocks: number of blocks allocated
- st_atime: time of last access
- st_mtime: time of last modification
- st_ctime: time of last status change

Syntax

Following is the syntax for fstat() method-

```
os.fstat(fd)
```

Parameters

fd - This is the file descriptor for which system information is to be returned.

Return Value

This method returns information about a file associated with the fd.

Example

The following example shows the usage of chdir() method.

```
#!/usr/bin/python3
```

```
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Now get  the touple
info = os.fstat(fd)

print ("File Info :", info)

# Now get uid of the file
print ("UID of the file :%d" % info.st_uid)

# Now get gid of the file
print ("GID of the file :%d" % info.st_gid)

# Close opened file
os.close( fd)
```

When we run the above program, it produces the following result-

```
File Info : os.stat_result(st_mode=33206, st_ino=2533274790483933, st_dev=1017554828, st_nlink=1, st_uid=0, st_gid=0, st_size=61, st_atime=1455562034, st_mtime=1455561637, st_ctime=1455561164)
UID of the file :0
GID of the file :0
```

os.fstatvfs() Method

Description

The method fstatvfs() returns information about the file system containing the file associated with file descriptor fd. This returns the following structure-

- f_bsize: file system block size
- f_frsize: fragment size
- f_blocks: size of fs in f_frsize units
- f_bfree: free blocks
- f_bavail: free blocks for non-root
- f_files: inodes

- f_ffree: free inodes
- f_favail: free inodes for non-root
- f_fsid: file system ID
- f_flag: mount flags
- f_namemax: maximum filename length

Syntax

Following is the syntax for fstatvfs() method-

```
os.fstatvfs(fd)
```

Parameters

fd - This is the file descriptor for which system information is to be returned.

Return Value

This method returns information about the file system containing the file associated.

Example

The following example shows the usage of fstatvfs() method.

```
#!/usr/bin/python3
import os, sys

# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )

# Now get  the touple
info = os.fstatvfs(fd)

print ("File Info :", info)

# Now get maximum filename length
print ("Maximum filename length :%d" % info.f_namemax:)

# Now get free blocks
```

```
print ("Free blocks :%d" % info.f_bfree)


# Close opened file
os.close( fd)
```

When we run the above program, it produces the following result-

```
File Info : (4096, 4096, 2621440L, 1113266L, 1113266L,
             8929602L, 8764252L, 8764252L, 0, 255)
Maximum filename length :255
Free blocks :1113266
```

os.fsync() Method

Description

The method fsync() forces write of file with file descriptor fd to disk. If you're starting with a Python file object f, first do f.flush(), and then do os.fsync(f.fileno()), to ensure that all internal buffers associated with f are written to disk.

Syntax

Following is the syntax for fsync() method-

```
os.fsync(fd)
```

Parameters

fd - This is the file descriptor for buffer sync is required.

Return Value

This method does not return any value.

Example

The following example shows the usage of fsync() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Write one string
```

```
line="this is test"
b=line.encode()
os.write(fd, b)
# Now you can use fsync() method.
# Infact here you would not be able to see its effect.
os.fsync(fd)
# Now read this file from the beginning
os.lseek(fd, 0, 0)
line = os.read(fd, 100)
b=line.decode()
print ("Read String is : ", b)
# Close opened file
os.close( fd )
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Read String is :  this is test
Closed the file successfully!!
```

os.ftruncate() Method

Description

The method ftruncate() truncates the file corresponding to file descriptor fd, so that it is at most length bytes in size.

Syntax

Following is the syntax for ftruncate() method-

```
os.ftruncate(fd, length)
```

Parameters

- **fd** - This is the file descriptor, which needs to be truncated.
- **length** - This is the length of the file where file needs to be truncated.

Return Value

This method does not return any value. Available on Unix like systems.

Example

The following example shows the usage of ftruncate() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )

# Write one string
os.write(fd, "This is test - This is test")

# Now you can use ftruncate() method.
os.ftruncate(fd, 10)

# Now read this file from the beginning.
os.lseek(fd, 0, 0)
str = os.read(fd, 100)
print ("Read String is : ", str)

# Close opened file
os.close( fd )
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Read String is :  This is te
Closed the file successfully!!
```

os.getcwd() Method

Description

The method getcwd() returns current working directory of a process.

Syntax

Following is the syntax for getcwd() method-

```
os.ggetcwd(path)
```

Parameters

NA

Return Value

This method returns the current working directory of a process.

Example

The following example shows the usage of getcwd() method-

```
#!/usr/bin/python3
import os, sys
# First go to the "/var/www/html" directory
os.chdir("/var/www/html" )

# Print current working directory
print ("Current working dir : %s" % os.getcwd())

# Now open a directory "/tmp"
fd = os.open( "/tmp", os.O_RDONLY )

# Use os.fchdir() method to change the dir
os.fchdir(fd)
# Print current working directory
print ("Current working dir : %s" % os.getcwd())
# Close opened directory.
os.close( fd )
```

When we run the above program, it produces the following result-

```
Current working dir : /var/www/html
Current working dir : /tmp
```

os.getcwdu() Method

Description

The method getcwdu() returns a unicode object representing the current working directory.

Syntax

Following is the syntax for getcwdu() method-

```
os.getcwdu()
```

Parameters

NA

Return Value

This method returns a unicode object representing the current working directory.

Example

The following example shows the usage of getcwdu() method.

```
#!/usr/bin/python3
import os, sys

# First go to the "/var/www/html" directory
os.chdir("/var/www/html" )

# Print current working directory
print ("Current working dir : %s" % os.getcwdu())

# Now open a directory "/tmp"
fd = os.open( "/tmp", os.O_RDONLY )

# Use os.fchdir() method to change the dir
os.fchdir(fd)

# Print current working directory
print ("Current working dir : %s" % os.getcwdu())

# Close opened directory.
os.close( fd )
```

When we run the above program, it produces the following result-

```
Current working dir : /var/www/html
Current working dir : /tmp
```

os.isatty() Method

Description

The method isatty()returns True if the file descriptor fd is open and connected to a tty(-like) device, else False.

Syntax

Following is the syntax for isatty() method-

```
os.isatty( fd )
```

Parameters

fd - This is the file descriptor for which association needs to be checked.

Return Value

This method returns True if the file descriptor fd is open and connected to a tty(-like) device, else False.

Example

The following example shows the usage of isatty() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Write one string
line="This is test"
b=line.encode()
os.write(fd, b)

# Now use isatty() to check the file.
ret = os.isatty(fd)

print ("Returned value is: ", ret)

# Close opened file
os.close( fd )
```

When we run the above program, it produces the following result-

```
Returned value is:  False
```

os.lchflags() Method

Description

The method lchflags() sets the flags of path to the numeric flags. This method does not follow symbolic links unlike chflags() method. As of Python 3.3, this is equivalent to os.chflags(path, flags, follow_symlinks=False).

Here, flags may take a combination (bitwise OR) of the following values (as defined in the stat module):

- UF_NODUMP: Do not dump the file.
- UF_IMMUTABLE: The file may not be changed.
- UF_APPEND: The file may only be appended to.
- UF_NOUNLINK: The file may not be renamed or deleted.
- UF_OPAQUE: The directory is opaque when viewed through a union stack.
- SF_ARCHIVED: The file may be archived.
- SF_IMMUTABLE: The file may not be changed.
- SF_APPEND: The file may only be appended to.
- SF_NOUNLINK: The file may not be renamed or deleted.
- SF_SNAPSHOT: The file is a snapshot file.

Note: This method has been introduced in Python 2.6

Syntax

Following is the syntax for lchflags() method-

```
os.lchflags(path, flags)
```

Parameters

- **path** - This is the file path for which flags to be set.
- **flags** - This could be a combination (bitwise OR) of the above defined flags values.

Return Value

This method does not return any value. Available on Unix like systems.

Example

The following example shows the usage of lchflags() method.

```
#!/usr/bin/python3

import os, sys

# Open a file
path = "/var/www/html/foo.txt"
fd = os.open( path, os.O_RDWR|os.O_CREAT )

# Close opened file
os.close( fd )

# Now change the file flag.
ret = os.lchflags(path, os.UF_IMMUTABLE )

print ("Changed file flag successfully!!")
```

When we run the above program, it produces the following result-

```
Changed file flag successfully!!
```

os.lchown() Method

Description

The method lchown() changes the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links. To leave one of the ids unchanged, set it to -1. As of Python 3.3, this is equivalent to os.chown(path, uid, gid, follow_symlinks=False).

Syntax

Following is the syntax for lchown() method-

```
os.lchown(path, uid, gid)
```

Parameters

- **path** - This is the file path for which ownership to be set.
- **uid** - This is the Owner ID to be set for the file.
- **gid** - This is the Group ID to be set for the file.

Return Value

This method does not return any value.

Example

The following example shows the usage of lchown() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
path = "/var/www/html/foo.txt"
fd = os.open( path, os.O_RDWR|os.O_CREAT )

# Close opened file
os.close( fd )

# Now change the file ownership.
# Set a file owner ID
os.lchown( path, 500, -1)

# Set a file group ID
os.lchown( path, -1, 500)

print ("Changed ownership successfully!!")
```

When we run above program, it produces following result-

```
Changed ownership successfully!!
```

os.link() Method

Description

The method link() creates a hard link pointing to src named dst. This method is very useful to create a copy of existing file.

Syntax

Following is the syntax for link() method-

```
os.link(src, dst)
```

Parameters

- **src** - This is the source file path for which hard link would be created.
- **dest** - This is the target file path where hard link would be created.

Return Value

This method does not return any value. Available on Unix, Windows.

Example

The following example shows the usage of link() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
path = "d:\\python3\\foo.txt"
fd = os.open( path, os.O_RDWR|os.O_CREAT )

# Close opened file
os.close( fd )

# Now create another copy of the above file.
dst = "d:\\tmp\\foo.txt"
os.link( path, dst)

print ("Created hard link successfully!!")
```

When we run the above program, it produces the following result-

```
Created hard link successfully!!
```

os.listdir() Method

Description

The method **listdir()** returns a list containing the names of the entries in the directory given by path. The list is in arbitrary order. It does not include the special entries '.' and '..' even if they are present in the directory.

path may be either of type str or of type bytes. If path is of type bytes, the filenames returned will also be of type bytes; in all other circumstances, they will be of type str.

Syntax

Following is the syntax for listdir() method-

```
os.listdir(path)
```

Parameters

path - This is the directory, which needs to be explored.

Return Value

This method returns a list containing the names of the entries in the directory given by path.

Example

The following example shows the usage of listdir() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
path = "d:\\tmp\\"
dirs = os.listdir( path )
# This would print all the files and directories
for file in dirs:
   print (file)
```

When we run the above program, it produces the following result-

```
Applicationdocs.docx
test.java
book.zip
foo.txt
Java Multiple Inheritance.htm
Java Multiple Inheritance_files
java.ppt
ParallelPortViewer
```

os.lseek() Method

Description

The method lseek() sets the current position of file descriptor fd to the given position pos, modified by how.

Syntax

Following is the syntax for lseek() method-

```
os.lseek(fd, pos, how)
```

Parameters

- **fd** - This is the file descriptor, which needs to be processed.

- **pos** - This is the position in the file with respect to given parameter how. You give os.SEEK_SET or 0 to set the position relative to the beginning of the file, os.SEEK_CUR or 1 to set it relative to the current position; os.SEEK_END or 2 to set it relative to the end of the file.

- **how** - This is the reference point with-in the file. os.SEEK_SET or 0 means beginning of the file, os.SEEK_CUR or 1 means the current position and os.SEEK_END or 2 means end of the file.

Defined **pos** constants

- os.SEEK_SET - 0
- os.SEEK_CUR - 1
- os.SEEK_END - 2

Return Value

This method does not return any value.

Example

The following example shows the usage of lseek() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Write one string
line="This is test"
b=line.encode()
os.write(fd, b)
# Now you can use fsync() method.
# Infact here you would not be able to see its effect.
os.fsync(fd)
# Now read this file from the beginning
os.lseek(fd, 0, 0)
```

```
line = os.read(fd, 100)
print ("Read String is : ", line.decode())

# Close opened file
os.close( fd )

print "Closed the file successfully!!"
```

When we run the above program, it produces the following result-

```
Read String is :  This is test
Closed the file successfully!!
```

os.lstat() Method

Description

The method lstat() is very similar to fstat() and returns a stat_result object containing the information about a file, but do not follow symbolic links. This is an alias for fstat() on platforms that do not support symbolic links, such as Windows.

Here is the structure returned by lstat method-

- st_dev: ID of device containing file
- st_ino: inode number
- st_mode: protection
- st_nlink: number of hard links
- st_uid: user ID of owner
- st_gid: group ID of owner
- st_rdev: device ID (if special file)
- st_size: total size, in bytes
- st_blksize: blocksize for filesystem I/O
- st_blocks: number of blocks allocated
- st_atime: time of last access
- st_mtime: time of last modification
- st_ctime: time of last status change

Syntax

Following is the syntax for lstat() method:

```
os.lstat(path)
```

Parameters

path - This is the file for which information would be returned.

Return Value

This method returns the information about a file.

Example

```
The following example shows the usage of lstat() method.
#!/usr/bin/python3
import os, sys
# Open a file
path = "d:\\python3\\foo.txt"
fd = os.open( path, os.O_RDWR|os.O_CREAT )
# Close opened file
os.close( fd )
# Now get  the touple
info = os.lstat(path)
print ("File Info :", info)
# Now get uid of the file
print ("UID of the file :%d" % info.st_uid)
# Now get gid of the file
print ("GID of the file :%d" % info.st_gid)
```

When we run the above program, it produces the following result-

```
File    Info    :    os.stat_result(st_mode=33206,    st_ino=281474976797706,
st_dev=1017554828,    st_nlink=2,    st_uid=0,    st_gid=0,    st_size=13,
st_atime=1455597777, st_mtime=1438077266, st_ctime=1455560006)
UID of the file :0
GID of the file :0
```

os.major() Method

Description

The method major() extracts the device major number from a raw device number (usually the st_dev or st_rdev field from stat).

Syntax

Following is the syntax for major() method-

```
os.major(device)
```

Parameters

device - This is a raw device number (usually the st_dev or st_rdev field from stat).

Return Value

This method returns the device major number.

Example

The following example shows the usage of major() method.

```
#!/usr/bin/python3
import os, sys
path = "/var/www/html/foo.txt"
# Now get  the touple
info = os.lstat(path)
# Get major and minor device number
major_dnum = os.major(info.st_dev)
minor_dnum = os.minor(info.st_dev)
print ("Major Device Number :", major_dnum)
print ("Minor Device Number :", minor_dnum)
```

When we run the above program, it produces the following result-

```
Major Device Number : 0
Minor Device Number : 103
```

os.makedev() Method

Description

The method makedev() composes a raw device number from the major and minor device numbers.

Syntax

Following is the syntax for makedev() method-

```
os.makedev(major, minor)
```

Parameters

- **major** - This is Major device number.
- **minor** - This is Minor device number.

Return Value

This method returns the device number.

Example

The following example shows the usage of makedev() method.

```
#!/usr/bin/python3
import os, sys
path = "/var/www/html/foo.txt"
# Now get  the touple
info = os.lstat(path)
# Get major and minor device number
major_dnum = os.major(info.st_dev)
minor_dnum = os.minor(info.st_dev)
print ("Major Device Number :", major_dnum)
print ("Minor Device Number :", minor_dnum)

# Make a device number
dev_num = os.makedev(major_dnum, minor_dnum)
print ("Device Number :", dev_num)
```

When we run the above program, it produces the following result-

```
Major Device Number : 0
Minor Device Number : 103
Device Number : 103
```

os.makedirs() Method

Description

The method makedirs() is recursive directory creation function. Like mkdir(), but makes all intermediate-level directories needed to contain the leaf directory.

The default mode is 0o777 (octal). On some systems, mode is ignored. Where it is used, the current umask value is first masked out.

If exist_ok is False (the default), an OSError is raised if the target directory already exists.

Syntax

Following is the syntax for makedirs() method-

```
os.makedirs(path[, mode])
```

Parameters

- **path** - This is the path, which needs to be created recursively.
- **mode** - This is the Mode of the directories to be given.

Return Value

This method does not return any value.

Example

The following example shows the usage of makedirs() method.

```
#!/usr/bin/python3
import os, sys
# Path to be created
path = "d:/tmp/home/monthly/daily"
os.makedirs( path, 493 ) #decimal equivalent of 0755 used on Windows
print ("Path is created")
```

When we run the above program, it produces the following result-

```
Path is created
```

os.minor() Method

Description

The method minor() extracts the device minor number from a raw device number (usually the st_dev or st_rdev field from stat).

Syntax

Following is the syntax for minor() method-

```
os.minor(device)
```

Parameters

device - This is a raw device number (usually the st_dev or st_rdev field from stat).

Return Value

This method returns the device minor number.

Example

The following example shows the usage of minor() method.

```
#!/usr/bin/python3
import os, sys
path = "/var/www/html/foo.txt"
# Now get the touple
info = os.lstat(path)
# Get major and minor device number
major_dnum = os.major(info.st_dev)
minor_dnum = os.minor(info.st_dev)

print ("Major Device Number :", major_dnum)
print ("Minor Device Number :", minor_dnum)
```

When we run the above program, it produces the following result-

```
Major Device Number : 0
Minor Device Number : 103
```

os.mkdir() Method

Description

The method mkdir() create a directory named path with numeric mode mode. The default mode is 0777 (octal). On some systems, mode is ignored. Where it is used, the current umask value is first masked out.

Syntax

Following is the syntax for mkdir() method-

```
os.mkdir(path[, mode])
```

Parameters

- **path** - This is the path, which needs to be created.
- **mode** - This is the mode of the directories to be given.

Return Value

This method does not return any value.

Example

The following example shows the usage of mkdir() method.

```
#!/usr/bin/python3
import os, sys
# Path to be created
path = "/tmp/home/monthly/daily/hourly"
os.mkdir( path, 0755 );
print "Path is created"
```

When we run the above program, it produces the following result-

```
Path is created
```

os.mkfifo() Method

Description

The method **mkfifo()** create a FIFO named path with numeric mode. The default mode is 0666 (octal).The current umask value is first masked out.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted

Syntax

Following is the syntax for mkfifo() method-

```
os.mkfifo(path[, mode])
```

Parameters

- **path** - This is the path, which needs to be created.
- **mode** - This is the mode of the named path to be given.

Return Value

This method does not return any value.

Example

The following example shows the usage of mkfifo() method.

```
# !/usr/bin/python3
import os, sys
# Path to be created
path = "/tmp/hourly"
os.mkfifo( path, 0644 )
print ("Path is created")
```

When we run the above program, it produces the following result-

```
Path is created
```

os.mknod() Method

Description

The method mknod() creates a filesystem node (file, device special file or named pipe) named filename.

Syntax

Following is the syntax for mknod() method-

```
os.mknod(filename[, mode=0600[, device=0]])
```

Parameters

- **filename** - This is the filesystem node to be created.
- **mode** - The mode specifies both the permissions to use and the type of node to be created combined (bitwise OR) with one of the values stat.S_IFREG, stat.S_IFCHR, stat.S_IFBLK, and stat.S_IFIFO. They can be ORed base don requirement.
- **device** - This is the device special file created and its optional to provide.

Return Value

This method does not return any value. Available on Unix like systems.

Example

The following example shows the usage of mknod() method.

```
# !/usr/bin/python3
import os
import stat
filename = '/tmp/tmpfile'
mode = 0600|stat.S_IRUSR
# filesystem node specified with different modes
os.mknod(filename, mode)
```

Let us compile and run the above program, this will create a simple file in /tmp directory with a name tmpfile:

```
-rw-------. 1 root   root         0 Apr 30 02:38 tmpfile
```

os.open() Method

Description

The method **open()** opens the file file and set various flags according to flags and possibly its mode according to mode.The default mode is 0777 (octal), and the current umask value is first masked out.

Syntax

Following is the syntax for open() method:

```
os.open(file, flags[, mode]);
```

Parameters

- **file** - File name to be opened.

- **flags** - The following constants are options for the flags. They can be combined using the bitwise OR operator |. Some of them are not available on all platforms.
  - os.O_RDONLY: open for reading only
  - os.O_WRONLY: open for writing only
  - os.O_RDWR : open for reading and writing
  - os.O_NONBLOCK: do not block on open

  - os.O_APPEND: append on each write
  - os.O_CREAT: create file if it does not exist
  - os.O_TRUNC: truncate size to 0
  - os.O_EXCL: error if create and file exists
  - os.O_SHLOCK: atomically obtain a shared lock
  - os.O_EXLOCK: atomically obtain an exclusive lock
  - os.O_DIRECT: eliminate or reduce cache effects
  - os.O_FSYNC : synchronous writes
  - os.O_NOFOLLOW: do not follow symlinks
- **mode** - This work in similar way as it works for chmod() method.

Return Value

This method returns the file descriptor for the newly opened file.

Example

The following example shows the usage of open() method.

```
#!/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "foo.txt", os.O_RDWR|os.O_CREAT )
# Write one string
line="this is test"
# string needs to be converted byte object
b=str.encode(line)
os.write(fd, b)
# Close opened file
os.close( fd)
print ("Closed the file successfully!!")
```

This would create given file foo.txt and then would write given content in that file and would produce the following result-

```
Closed the file successfully!!
```

os.openpty() Method

Description

The method **openpty()** opens a pseudo-terminal pair and returns a pair of file descriptors(master,slave) for the pty & the tty respectively.

The new file descriptors are non-inheritable. For a (slightly) more portable approach, use the pty module.

Syntax

Following is the syntax for openpty() method-

```
os.openpty()
```

Parameters

NA

Return Value

This method returns a pair of file descriptors i.e., master and slave.

Example

```
The following example shows the usage of openpty() method.
# !/usr/bin/python3
import os
# master for pty, slave for tty
m,s = os.openpty()
print (m)
print (s)
# showing terminal name
s = os.ttyname(s)
print (m)
print( s)
```

When we run the above program, it produces the following result-

```
3
4
3
/dev/pty0
```

os.pathconf() Method

Description

The method pathconf() returns system configuration information relevant to a named file.

Syntax

Following is the syntax for pathconf() method-

```
os.pathconf(path, name)
```

Parameters

- **path** - This is the file path.
- **name** - This specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). The names known to the host operating system are given in the os.pathconf_names dictionary.

Return Value

This method returns system configuration information of a file. Available on Unix like systems.

Example

The following example shows the usage of pathconf() method.

```
#!/usr/bin/python3
import os, sys
print ("%s" % os.pathconf_names)
# Retrieve maximum length of a filename
no = os.pathconf('a2.py', 'PC_NAME_MAX')
print ("Maximum length of a filename :%d" % no)
# Retrieve file size
no = os.pathconf('a2.py', 'PC_FILESIZEBITS')
print ("file size in bits  :%d" % no)
```

When we run the above program, it produces the following result-

```
{'PC_MAX_INPUT': 2, 'PC_VDISABLE': 8, 'PC_SYNC_IO': 9,
'PC_SOCK_MAXBUF': 12, 'PC_NAME_MAX': 3, 'PC_MAX_CANON': 1,
'PC_PRIO_IO': 11, 'PC_CHOWN_RESTRICTED': 6, 'PC_ASYNC_IO': 10,
'PC_NO_TRUNC': 7, 'PC_FILESIZEBITS': 13, 'PC_LINK_MAX': 0,
'PC_PIPE_BUF': 5, 'PC_PATH_MAX': 4}
 Maximum length of a filename :255
```

```
file size in bits : 64
```

os.pipe() Method

Description

The method pipe() creates a pipe and returns a pair of file descriptors (r, w) usable for reading and writing, respectively

Syntax

Following is the syntax for pipe() method-

```
os.pipe()
```

Parameters

NA

Return Value

This method returns a pair of file descriptors.

Example

The following example shows the usage of pipe() method.

```
#!/usr/bin/python3
import os, sys

print ("The child will write text to a pipe and ")
print ("the parent will read the text written by child...")

# file descriptors r, w for reading and writing
r, w = os.pipe()

processid = os.fork()
if processid:
    # This is the parent process
    # Closes file descriptor w
    os.close(w)
    r = os.fdopen(r)
    print ("Parent reading")
```

```
    str = r.read()
    print ("text =", str   )
    sys.exit(0)
else:
    # This is the child process
    os.close(r)
    w = os.fdopen(w, 'w')
    print ("Child writing")
    w.write("Text written by child...")
    w.close()
    print ("Child closing")
    sys.exit(0)
```

When we run the above program, it produces the following result-

```
The child will write text to a pipe and
the parent will read the text written by child...
Parent reading
Child writing
Child closing
text = Text written by child...
```

os.popen() Method

Description

The method popen() opens a pipe to or from command.The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is 'r' (default) or 'w'.The bufsize argument has the same meaning as in open() function.

Syntax

Following is the syntax for popen() method-

```
os.popen(command[, mode[, bufsize]])
```

Parameters

- **command** - This is command used.
- **mode** - This is the Mode can be 'r'(default) or 'w'.

- **bufsize** - If the buffering value is set to 0, no buffering will take place. If the buffering value is 1, line buffering will be performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action will be performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

Return Value

This method returns an open file object connected to the pipe.

Example

The following example shows the usage of popen() method.

```
# !/usr/bin/python3
import os, sys
# using command mkdir
a = 'mkdir nwdir'

b = os.popen(a,'r',1)
print b
```

When we run the above program, it produces the following result-

```
open file 'mkdir nwdir', mode 'r' at 0x81614d0
```

os.read() Method

Description

The method read() reads at most n bytes from file desciptor fd, return a string containing the bytes read. If the end of file referred to by fd has been reached, an empty string is returned.

Note: This function is intended for low-level I/O and must be applied to a file descriptor as returned by os.open() or pipe(). To read a "file object" returned by the built-in function open() or by popen() or fdopen(), or sys.stdin, use its read() or readline() methods.

Syntax

Following is the syntax for read() method-

```
os.read(fd,n)
```

Parameters

- **fd** - This is the file descriptor of the file.

- **n** - These are n bytes from file descriptor fd.

Return Value

This method returns a string containing the bytes read.

Example

The following example shows the usage of read() method.

```
# !/usr/bin/python3
import os, sys
# Open a file
fd = os.open("foo.txt",os.O_RDWR)
# Reading text
ret = os.read(fd,12)
print (ret.decode())
# Close opened file
os.close(fd)
print ("Closed the file successfully!!")
```

Let us compile and run the above program, this will print the contents of file foo.txt-

```
This is test
Closed the file successfully!!
```

os.readlink() Method

Description

The method **readlink()** returns a string representing the path to which the symbolic link points. It may return an absolute or relative pathname.

Syntax

Following is the syntax for readlink() method-

```
os.readlink(path)
```

Parameters

path - This is the path or symblic link for which we are going to find source of the link.

Return Value

This method return a string representing the path to which the symbolic link points.

Example

The following example shows the usage of readlink() method.

```
# !/usr/bin/python3
import os
src = 'd://tmp//python3'
dst = 'd://tmp//python2'
# This creates a symbolic link on python in tmp directory
os.symlink(src, dst)
# Now let us use readlink to display the source of the link.
path = os.readlink( dst )
print (path)
```

Let us compile and run the above program. This will create a symblic link to d:\tmp\python3 and later it will read the source of the symbolic link using readlink() call. This is an example on Windows platform and needs administrator privilege to run. Before running this program make sure you do not have d:\tmp\python2 already available.

```
d:\tmp\python2
```

os.remove() Method

Description

The method remove() removes the file path. If the path is a directory, OSError is raised.

Syntax

Following is the syntax for remove() method-

```
os.remove(path)
```

Parameters

path - This is the path, which is to be removed.

Return Value

This method does not return any value.

Example

The following example shows the usage of remove() method.

```
# !/usr/bin/python3
import os, sys
os.chdir("d:\\tmp")
```

```
# listing directories
print ("The dir is: %s" %os.listdir(os.getcwd()))

# removing
os.remove("test.java")

# listing directories after removing path
print ("The dir after removal of path : %s" %os.listdir(os.getcwd()))
```

When we run above program, it produces following result-

```
The dir is: ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'home', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'ParallelPortViewer', 'test.java']
The dir after removal of path : ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'home', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'ParallelPortViewer']
```

os.removedirs() Method

Description

The method removedirs() removes dirs recursively. If the leaf directory is succesfully removed, removedirs tries to successively remove every parent directory displayed in path. Raises OSError if the leaf directory could not be successfully removed.

Syntax

Following is the syntax for removedirs() method-

```
os.removedirs(path)
```

Parameters

path - This is the path of the directory, which needs to be removed.

Return Value

This method does not return any value.

Example

The following example shows the usage of removedirs() method.

```
# !/usr/bin/python3
import os, sys
```

```
os.chdir("d:\\tmp")
# listing directories
print ("The dir is: %s" %os.listdir(os.getcwd()))
# removing
os.removedirs("home\\monthly\\daily")
# listing directories after removing directory
print ("The dir after removal is:" %os.listdir(os.getcwd()))
```

When we run the above program, it produces the following result-

```
The dir is: ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'home', 'Java
Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt',
'ParallelPortViewer']
The dir after removal is:
['Applicationdocs.docx', 'book.zip', 'foo.txt', 'Java Multiple Inheritance.htm',
'Java Multiple Inheritance_files', 'java.ppt', 'ParallelPortViewer']
```

os.rename() Method

Description

The method **rename()** renames the file or directory src to dst. If dst is a file or directory(already present), OSError will be raised.

Syntax

Following is the syntax for rename() method-

```
os.rename(src, dst)
```

Parameters

- **src** - This is the actual name of the file or directory.
- **dst** - This is the new name of the file or directory.

Return Value

This method does not return any value.

Example

The following example shows the usage of rename() method.

```
# !/usr/bin/python3
import os, sys
```

```
os.chdir("d:\\tmp")
# listing directories
print ("The dir is: %s"%os.listdir(os.getcwd()))
# renaming directory ''tutorialsdir"
os.rename("python3","python2")
print ("Successfully renamed.")
# listing directories after renaming "python3"
print ("the dir is: %s" %os.listdir(os.getcwd()))
```

When we run the above program, it produces the following result-

```
The dir is: ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'Python3']
Successfully renamed.
the dir is: ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'python2']
```

os.renames() Method

Description

The method renames() is recursive directory or file renaming function. It does the same functioning as os.rename(), but it also moves a file to a directory, or a whole tree of directories, that do not exist.

Syntax

Following is the syntax for renames() method:

```
os.renames(old, new)
```

Parameters

- **old** - This is the actual name of the file or directory to be renamed.
- **new** - This is the new name of the file or directory. It can even include a file to a directory, or a whole tree of directories, that do not exist.

Return Value

This method does not return any value.

Example

The following example shows the usage of renames() method.

```
# !/usr/bin/python3
import os, sys
os.chdir("d:\\tmp")
print ("Current directory is: %s" %os.getcwd())
# listing directories
print ("The dir is: %s"%os.listdir(os.getcwd()))
# renaming file "aa1.txt"
os.renames("foo.txt","newdir/foonew.txt")
print ("Successfully renamed.")
# listing directories after renaming and moving "foo.txt"
print ("The dir is: %s" %os.listdir(os.getcwd()))
os.chdir("newdir")
print ("The dir is: %s" %os.listdir(os.getcwd()))
```

When we run the above program, it produces the following result-

```
Current directory is: d:\tmp
The  dir  is:  ['Applicationdocs.docx',  'book.zip',  'foo.txt',  'Java  Multiple
Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'python2']
Successfully renamed.
The dir is: ['Applicationdocs.docx', 'book.zip', 'Java Multiple Inheritance.htm',
'Java Multiple Inheritance_files', 'java.ppt', 'newdir', 'python2']
The file foo.txt is not visible here, as it is been moved to newdir and renamed
as foonew.txt. The directory newdir and its contents are shown below:
The dir is: ['foonew.txt']
```

os.renames() Method

Description

The method renames() is recursive directory or file renaming function. It does the same functioning as os.rename(), but it also moves a file to a directory, or a whole tree of directories, that do not exist.

Syntax

Following is the syntax for renames() method-

```
os.renames(old, new)
```

Parameters

- **old** - This is the actual name of the file or directory to be renamed.

- **new** - This is the new name of the file or directory.It can even include a file to a directory, or a whole tree of directories, that do not exist.

Return Value

This method does not return any value.

Example

The following example shows the usage of renames() method.

```
# !/usr/bin/python3
import os, sys
os.chdir("d:\\tmp")
print ("Current directory is: %s" %os.getcwd())
# listing directories
print ("The dir is: %s"%os.listdir(os.getcwd()))
# renaming file "aa1.txt"
os.renames("foo.txt","newdir/foonew.txt")

print ("Successfully renamed.")
# listing directories after renaming and moving "foo.txt"
print ("The dir is: %s" %os.listdir(os.getcwd()))
os.chdir("newdir")
print ("The dir is: %s" %os.listdir(os.getcwd()))
```

When we run the above program, it produces the following result-

```
Current directory is: d:\tmp
The dir is: ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'python2']
Successfully renamed.
The dir is: ['Applicationdocs.docx', 'book.zip', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'newdir', 'python2']
The file foo.txt is not visible here, as it is been moved to newdir and renamed as foonew.txt. The directory newdir and its contents are shown below:
The dir is: ['foonew.txt']
```

os.rmdir() Method

Description

The method **rmdir()** removes the directory path. It works only when the directory is empty, else OSError is raised.

Syntax

Following is the syntax for rmdir() method-

```
os.rmdir(path)
```

Parameters

path - This is the path of the directory, which needs to be removed.

Return Value

This method does not return any value.

Example

```
The following example shows the usage of rmdir() method.
# !/usr/bin/python3
import os, sys
os.chdir("d:\\tmp")
# listing directories
print ("the dir is: %s" %os.listdir(os.getcwd()))

# removing path
os.rmdir("newdir")

# listing directories after removing directory path
print ("the dir is:" %os.listdir(os.getcwd()))
```

When we run the above program, it produces the following result-

```
the dir is: ['Applicationdocs.docx', 'book.zip', 'Java Multiple Inheritance.htm',
'Java Multiple Inheritance_files', 'java.ppt', 'newdir', 'python2']
Traceback (most recent call last):
  File "test.py", line 8, in
    os.rmdir("newdir")
OSError: [WinError 145] The directory is not empty: 'newdir'
The error is coming as 'newdir' directory is not empty. If 'newdir' is an empty
directory, then this would produce following result:
```

```
the dir is: ['Applicationdocs.docx', 'book.zip', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'newdir', 'python2']

the dir is: ['Applicationdocs.docx', 'book.zip', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'python2']
```

os.stat() Method

Description

The method **stat()** performs a stat system call on the given path.

Syntax

Following is the syntax for stat() method-

```
os.stat(path)
```

Parameters

path - This is the path, whose stat information is required.

Return Value

Here is the list of members of stat structure-

- st_mode: protection bits.
- st_ino: inode number.
- st_dev: device.
- st_nlink: number of hard links.
- st_uid: user id of owner.
- st_gid: group id of owner.
- st_size: size of file, in bytes.
- st_atime: time of most recent access.
- st_mtime: time of most recent content modification.
- st_ctime: time of most recent metadata change.

Example

The following example shows the usage of stat() method.

```
# !/usr/bin/python3
import os, sys
# showing stat information of file "foo.txt"
statinfo = os.stat('foo.txt')
```

```
print (statinfo)
```

When we run the above program, it produces the following result-

```
os.stat_result(st_mode=33206,    st_ino=281474976797706,    st_dev=1017554828,
st_nlink=1,    st_uid=0,    st_gid=0,    st_size=13,    st_atime=1455649253,
st_mtime=1438077266, st_ctime=1455560006)
```

os.stat_float_times() Method

Description

The method stat_float_times() determines whether stat_result represents time stamps as float objects.

Syntax

Following is the syntax for stat_float_times() method-

```
os.stat_float_times([newvalue])
```

Parameters

newvalue - If newvalue is True, future calls to stat() return floats, if it is False, future call on stat returns ints. If newvalue is not mentioned, it returns the current settings.

Return Value

This method returns either True or False.

Example

The following example shows the usage of stat_float_times() method.

```
#!/usr/bin/python3
import os, sys
# Stat information
statinfo = os.stat('a2.py')

print (statinfo)
statinfo = os.stat_float_times()
print (statinfo)
```

When we run the above program, it produces the following result-

```
os.stat_result(st_mode=33206,     st_ino=562949953508433,     st_dev=1017554828,
st_nlink=1,     st_uid=0,     st_gid=0,     st_size=27,     st_atime=1455597032,
st_mtime=1455597032, st_ctime=1455562995)
True
```

os.statvfs() Method

Description

The method statvfs() perform a statvfs system call on the given path.

Syntax

Following is the syntax for statvfs() method-

```
os.statvfs(path)
```

Parameters

path - This is the path, whose statvfs information is required.

Return Value

Here is the list of members of statvfs structure-

- f_bsize: preferred file system block size.
- f_frsize: fundamental file system block size.
- f_blocks: total number of blocks in the filesystem.
- f_bfree: total number of free blocks.
- f_bavail: free blocks available to non-super user.
- f_files: total number of file nodes.
- f_ffree: total number of free file nodes.
- f_favail: free nodes available to non-super user.
- f_flag: system dependent.
- f_namemax: maximum file name length.

Example

The following example shows the usage of statvfs() method. Availabe on Unix like systems-

```
# !/usr/bin/python3
import os, sys
# showing statvfs information of file "a1.py"
stinfo = os.statvfs('a1.py')
```

```
print (stinfo)
```

When we run the above program, it produces the following result-

```
posix.statvfs_result(f_bsize=4096, f_frsize=4096, f_blocks=1909350L,
f_bfree=1491513L,
f_bavail=1394521L, f_files=971520L, f_ffree=883302L, f_fvail=883302L, f_flag=0,
f_namemax=255)
```

os.symlink() Method

Description

The method symlink() creates a symbolic link dst pointing to src.

Syntax

Following is the syntax for symlink() method-

```
os.symlink(src, dst)
```

Parameters

- **src** - This is the source.
- **dest** - This is the destination, which did not exist previously.

Return Value

This method does not return any value.

Example

The following example shows the usage of symlink() method-

```
#!/usr/bin/python3
import os
src = '/usr/bin/python3'
dst = '/tmp/python'
# This creates a symbolic link on python in tmp directory
os.symlink(src, dst)
print "symlink created"
```

Let us compile and run the above program, this will create a symbolic link in /tmp directory which will be as follows-

```
lrwxrwxrwx. 1 root root 15 Apr 30 03:00 python -> /usr/bin/python3
```

os.tcgetpgrp() Method

Description

The method tcgetpgrp() returns the process group associated with the terminal given by fd (an open file descriptor as returned by os.open())

Syntax

Following is the syntax for tcgetpgrp() method-

```
os.tcgetpgrp(fd)
```

Parameters

fd - This is the file descriptor.

Return Value

This method returns the process group.

Example

The following example shows the usage of tcgetpgrp() method-

```
# !/usr/bin/python3
import os, sys
# Showing current directory
print ("Current working dir :%s" %os.getcwd())
# Changing dir to /dev/tty
fd = os.open("/dev/tty",os.O_RDONLY)
f = os.tcgetpgrp(fd)
# Showing the process group
print ("the process group associated is: ")
print (f)

os.close(fd)
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Current working dir is :/tmp
the process group associated is:
```

```
2670
Closed the file successfully!!
```

os.tcsetpgrp() Method

Description

The method tcsetpgrp() sets the process group associated with the terminal given by fd (an open file descriptor as returned by os.open()) to pg.

Syntax

Following is the syntax for tcsetpgrp() method-

```
os.tcsetpgrp(fd, pg)
```

Parameters

- **fd** - This is the file descriptor.
- **pg** - This set the process group to pg.

Return Value

This method does not return any value.

Example

The following example shows the usage of tcsetpgrp() method.

```
# !/usr/bin/python3
import os, sys
# Showing current directory
print ("Current working dir :%s" %os.getcwd())

# Changing dir to /dev/tty
fd = os.open("/dev/tty",os.O_RDONLY)

f = os.tcgetpgrp(fd)

# Showing the process group
print ("the process group associated is: ")
print (f)
```

```
# Setting the process group
os.tcsetpgrp(fd,2672)
print ("done")

os.close(fd)
print "Closed the file successfully!!"
```

When we run the above program, it produces the following result-

```
Current working dir is :/tmp
the process group associated is:
2672
done
Closed the file successfully!!
```

os.tempnam() Method

Description

The method tempnam() returns a unique path name that is reasonable for creating a temporary file.

Syntax

Following is the syntax for tempnam() method-

```
os.tempnam(dir, prefix)
```

Parameters

- **dir** - This is the dir where the temporary filename will be created.
- **prefix** - This is the prefix of the generated temporary filename.

Return Value

This method returns a unique path.

Example

The following example shows the usage of tempnam() method.

```
# !/usr/bin/python3
import os, sys
# prefix is tuts1 of the generated file
tmpfn = os.tempnam('/tmp/tutorialsdir,'tuts1')
```

```
print "This is the unique path:"
print tmpfn
```

When we run the above program, it produces the following result-

```
This is the unique path:
/tmp/tutorialsdir/tuts1IbAco8
```

os.tmpfile() Method

Description

The method tmpfile() returns a new temporary file object opened in update mode (w+b). The file has no directory entries associated with it and will be deleted automatically once there are no file descriptors.

Syntax

Following is the syntax for tmpfile() method-

```
os.tmpfile
```

Parameters

NA

Return Value

This method returns a new temporary file object.

Example

The following example shows the usage of tmpfile() method.

```
# !/usr/bin/python3
import os
# The file has no directory entries associated with it and will be
# deleted automatically once there are no file descriptors.
tmpfile = os.tmpfile()
tmpfile.write('Temporary newfile is here.....')
tmpfile.seek(0)

print tmpfile.read()
```

```
tmpfile.close
```

When we run the above program, it produces the following result-

```
Temporary newfile is here.....
```

os.tmpnam() Method

Description

The method **tmpnam()** returns a unique path name that is reasonable for creating a temporary file.

Syntax

Following is the syntax for tmpnam() method-

```
os.tmpnam()
```

Parameters

NA

Return Value

This method returns a unique path name.

Example

```
The following example shows the usage of tmpnam() method.
# !/usr/bin/python3
import os, sys
# Temporary file generated in current directory
tmpfn = os.tmpnam()
print "This is the unique path:"
print tmpfn
```

When we run the above program, it produces the following result-

```
This is the unique path:
/tmp/fileUFojpd
```

os.ttyname() Method

Description

The method **ttyname()** returns a string, which specifies the terminal device associated with fd. If fd is not associated with a terminal device, an exception is raised.

Syntax

Following is the syntax for ttyname() method-

```
os.ttyname(fd)
```

Parameters

fd - This is the file descriptor.

Return Value

This method returns a string which specifies the terminal device. Available on Unix like Systems.

Example

The following example shows the usage of ttyname() method.

```
# !/usr/bin/python33
import os, sys

# Showing current directory
print ("Current working dir :%s" %os.getcwd())

# Changing dir to /dev/tty
fd = os.open("/dev/tty",os.O_RDONLY)

p = os.ttyname(fd)
print ("the terminal device associated is: ")
print p
print ("done!!")

os.close(fd)
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
Current working dir is :/tmp
the terminal device associated is:
/dev/tty
```

```
done!!
Closed the file successfully!!
```

os.unlink() Method

Description

The method **unlink()** removes (deletes) the file path. If the path is a directory, OSError is raised. This function is identical to the remove() mehod; the unlink name is its traditional Unix name.

Syntax

Following is the syntax for unlink() method-

```
os.unlink(path)
```

Parameters

path - This is the path, which is to be removed.

Return Value

This method does not return any value.

Example

The following example shows the usage of unlink() method.

```
# !/usr/bin/python3
import os, sys
os.chdir("d:\\tmp")
# listing directories
print ("The dir is: %s" %os.listdir(os.getcwd()))
os.unlink("foo.txt")
# listing directories after removing path
print ("The dir after removal of path : %s" %os.listdir(os.getcwd()))
```

When we run the above program, it produces the following result-

```
The dir is: ['Applicationdocs.docx', 'book.zip', 'foo.txt', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'python2']
The dir after removal of path : ['Applicationdocs.docx', 'book.zip', 'Java Multiple Inheritance.htm', 'Java Multiple Inheritance_files', 'java.ppt', 'python2']
```

os.utime() Method

Description

The method utime() sets the access and modified times of the file specified by path.

Syntax

Following is the syntax for utime() method-

```
os.utime(path, times)
```

Parameters

- **path** - This is the path of the file.
- **times** - This is the file access and modified time. If times is none, then the file access and modified times are set to the current time. The parameter times consists of row in the form of (atime, mtime) i.e (accesstime, modifiedtime).

Return Value

This method does not return any value.

Example

The following example shows the usage of utime() method.

```
# !/usr/bin/python3
import os, sys, time
os.chdir("d:\\tmp")
# Showing stat information of file
stinfo = os.stat('foo.txt')
print (stinfo)
# Using os.stat to recieve atime and mtime of file
print ("access time of foo.txt: %s" %stinfo.st_atime)
print ("modified time of foo.txt: %s" %stinfo.st_mtime)
print (time.asctime( time.localtime(stinfo.st_atime)))
# Modifying atime and mtime
os.utime("foo.txt",(1330712280, 1330712292))
print ("after modification")
print (time.asctime( time.localtime(stinfo.st_atime)))
print ("done!!")
```

When we run the above program, it produces the following result-

```
os.stat_result(st_mode=33206,    st_ino=1688849860351098,    st_dev=1017554828,
st_nlink=1,     st_uid=0,     st_gid=0,     st_size=0,     st_atime=1455684273,
st_mtime=1455684273, st_ctime=1455683589)

access time of foo.txt: 1455684273.84375

modified time of foo.txt: 1455684273.84375

Wed Feb 17 10:14:33 2016

after modification

Fri Mar  2 23:48:00 2012

done!!
```

os.walk() Method

Description

The method **walk()** generates the file names in a directory tree by walking the tree either top-down or bottom-up.

Syntax

Following is the syntax for the walk() method-

```
os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])
```

Parameters

- **top** - Each directory rooted at directory, yields 3-tuples, i.e., (dirpath, dirnames, filenames)
- **topdown** - If optional argument topdown is True or not specified, directories are scanned from top-down. If topdown is set to False, directories are scanned from bottom-up.
- **onerror** - This can show error to continue with the walk, or raise the exception to abort the walk.
- **followlinks** - This visits directories pointed to by symlinks, if set to true.

Return Value

This method does not return any value.

Example

The following example shows the usage of walk() method.

```
# !/usr/bin/python3

import os
```

```
os.chdir("d:\\tmp")
for root, dirs, files in os.walk(".", topdown=False):
    for name in files:
        print(os.path.join(root, name))
    for name in dirs:
        print(os.path.join(root, name))
```

Let us compile and run the above program. This will scan all the directories and subdirectories bottom-to-up.

```
.\python2\testdir\Readme_files\Lpt_Port_Config.gif
.\python2\testdir\Readme_files\ParallelPortViever.gif
.\python2\testdir\Readme_files\softcollection.css
.\python2\testdir\Readme_files\Thumbs.db
.\python2\testdir\Readme_files\Yellov_Ball.gif
.\python2\testdir\Readme.htm
.\python2\testdir\Readme_files
.\python2\testdir
.\Applicationdocs.docx
.\book.zip
.\foo.txt
.\java.ppt
.\python2
```

If you will change the value of topdown to True, then it will give you the following result-

```
.\Applicationdocs.docx
.\book.zip
.\foo.txt
.\java.ppt
.\python2
.\python2\testdir
.\python2\testdir\Readme.htm
.\python2\testdir\Readme_files
.\python2\testdir\Readme_files\Lpt_Port_Config.gif
.\python2\testdir\Readme_files\ParallelPortViever.gif
.\python2\testdir\Readme_files\softcollection.css
.\python2\testdir\Readme_files\Thumbs.db
```

```
.\python2\testdir\Readme_files\Yellov_Ball.gif
```

os.write() Method

Description

The method **write()** writes the string str to file descriptor fd. It returns the number of bytes actually written.

Syntax

Following is the syntax for write() method-

```
os.write(fd, str)
```

Parameters

- **fd** - This is the file descriptor.
- **str** - This is the string to be written.

Return Value

This method returns the number of bytes actually written.

Example

The following example shows the usage of the write() method-

```
# !/usr/bin/python3
import os, sys
# Open a file
fd = os.open( "f1.txt", os.O_RDWR|os.O_CREAT )
# Write one string
line="this is test"
# string needs to be converted byte object
b=str.encode(line)
ret=os.write(fd, b)
# ret consists of number of bytes written to f1.txt
print ("the number of bytes written: ", ret)
# Close opened file
os.close( fd)
print ("Closed the file successfully!!")
```

When we run the above program, it produces the following result-

```
the number of bytes written: 12
Closed the file successfully!!
```

18. Python 3 – Exceptions Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them-

- **Exception Handling.**
- **Assertions.**

Standard Exceptions

Here is a list of Standard Exceptions available in Python.

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.

EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

- The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

- Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises anAssertionError exception.

The syntax for assert is –

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception, using the try-except statement. If they are not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a given temperature from degrees Kelvin to degrees Fahrenheit. Since 0° K is as cold as it gets, the function bails out if it sees a negative temperature –

```
#!/usr/bin/python3
def KelvinToFahrenheit(Temperature):
    assert (Temperature >= 0),"Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32

print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

When the above code is executed, it produces the following result-

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an Exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of try....except...else blocks-

```
try:
   You do your operations here
   ......................
```

```
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax-

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example

This example opens a file, writes content in the file and comes out gracefully because there is no problem at all.

```
#!/usr/bin/python3

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
```

This produces the following result-

```
Written content in the file successfully
```

Example

This example tries to open a file where you do not have the write permission, so it raises an exception-

```
#!/usr/bin/python3
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
```

This produces the following result-

```
Error: can't find file or read data
```

The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows-

```
try:
   You do your operations here
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions

You can also use the same *except* statement to handle multiple exceptions as follows-

```
try:
   You do your operations here
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
```

```
else:
   If there is no exception then execute this block.
```

The try-finally Clause

You can use a **finally:** block along with a **try:** block. The **finally:** block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this-

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

Note: You can provide except clause(s), or a finally clause, but not both. You cannot use *else* clause as well along with a finally clause.

Example

```
#!/usr/bin/python3

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print ("Error: can\'t find file or read data")
   fh.close()
```

If you do not have permission to open the file in writing mode, then this will produce the following result-

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows-

```
#!/usr/bin/python3

try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
```

```
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print ("Error: can\'t find file or read data")
```

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows-

```
try:
   You do your operations here
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception-

```
#!/usr/bin/python3

# Define a function here.
def temp_convert(var):
try:
returnint(var)
except ValueError as Argument:
print("The argument does not contain numbers\n",Argument)

# Call above function here.
```

```
temp_convert("xyz")
```

This produces the following result-

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

Raising an Exception

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows-

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows-

```
def functionName( level ):
    if level <1:
        raise Exception(level)
        # The code below to this would not be executed
        # if we raise the exception
    return level
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either as a class object or a simple string. For example, to capture the above exception, we must write the except clause as follows-

```
try:
   Business Logic here...
except Exception as e:
   Exception handling here using e.args...
```

```
else:
   Rest of the code here...
```

The following example illustrates the use of raising an exception-

```
#!/usr/bin/python3
def functionName( level ):
    if level <1:
        raise Exception(level)
        # The code below to this would not be executed
        # if we raise the exception
    return level

try:
    l=functionName(-10)
    print ("level=",l)
except Exception as e:
    print ("error in level argument",e.args[0])
```

This will produce the following result-

```
error in level argument -10
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable **e** is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg
```

So once you have defined the above class, you can raise the exception as follows-

```
try:
   raise Networkerror("Bad hostname")
except Networkerror,e:
   print e.args
```

Python 3 – Advanced Tutorial

19. Python 3 – Object Oriented

Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support.

If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts.

However, here is a small introduction of Object-Oriented Programming (OOP) to help you.

Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

- **Class variable:** A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.

- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.

- **Instance:** An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.

- **Instantiation:** The creation of an instance of a class.

- **Method :** A special kind of function that is defined in a class definition.

- **Object:** A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

- **Operator overloading:** The assignment of more than one function to a particular operator.

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows-

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed **via*ClassName.__doc__*.**
- The ***class_suite*** consists of all the component statements defining class members, data attributes and functions.

Example

Following is an example of a simple Python class-

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)
```

- The variable *empCount* is a class variable whose value is shared among all the instances of **a** in this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Creating Instance Objects

To create instances of a class, you call the class using class name and pass in whatever arguments its *__init__* method accepts.

```
This would create first object of Employee class
emp1 = Employee("Zara", 2000)

This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows-

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

Now, putting all the concepts together-

```
#!/usr/bin/python3

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)
#This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
#This would create second object of Employee class"
```

```
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

When the above code is executed, it produces the following result-

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

You can add, remove, or modify attributes of classes and objects at any time-

```
emp1.salary = 7000  # Add an 'salary' attribute.
emp1.name = 'xyz'  # Modify 'age' attribute.
del emp1.salary  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions-

- The **getattr(obj, name[, default])**: to access the attribute of object.
- The **hasattr(obj,name)**: to check if an attribute exists or not.
- The **setattr(obj,name,value)**: to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)**: to delete an attribute.

```
hasattr(emp1, 'salary')    # Returns true if 'salary' attribute exists
getattr(emp1, 'salary')    # Returns value of 'salary' attribute
setattr(emp1, 'salary', 7000) # Set attribute 'age' at 8
delattr(emp1, 'salary')    # Delete attribute 'age'
```

Built-In Class Attributes

Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute –

- **__dict__:** Dictionary containing the class's namespace.
- **__doc__:** Class documentation string or none, if undefined.
- **__name__:** Class name.
- **__module__:** Module name in which the class is defined. This attribute is "__main__" in interactive mode.

- **__bases__:** A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes-

```
#!/usr/bin/python3

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)

emp1 = Employee("Zara", 2000)
emp2 = Employee("Manni", 5000)
print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__ )
```

When the above code is executed, it produces the following result-

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (,)
Employee.__dict__: {'displayCount': , '__module__': '__main__', '__doc__': 'Common base class for all employees', 'empCount': 2, '__init__': , 'displayEmployee': , '__weakref__': , '__dict__': }
```

Destroying Objects (Garbage Collection)

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed as Garbage Collection.

Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero. An object's reference count changes as the number of aliases that point to it changes.

An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it is deleted with *del*, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python collects it automatically.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. However, a class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non-memory resources used by an instance.

Example

This __del__() destructor prints the class name of an instance that is about to be destroyed.

```
#!/usr/bin/python3

class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print (class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
```

```
print (id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts)
del pt1
del pt2
del pt3
```

When the above code is executed, it produces the following result-

```
3083401324 3083401324 3083401324
Point destroyed
```

Note: Ideally, you should define your classes in a separate file, then you should import them in your main program file using *import* statement.

In the above example, assuming definition of a Point class is contained in *point.py* and there is no other executable code in it.

```
#!/usr/bin/python3
import point
p1=point.Point()
```

Class Inheritance

Instead of starting from a scratch, you can create a class by deriving it from a pre-existing class by listing the parent class in parentheses after the new class name.

The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name –

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

Example

```
#!/usr/bin/python3
class Parent:        # define parent class
   parentAttr = 100
   def __init__(self):
```

```
      print ("Calling parent constructor")
   def parentMethod(self):
      print ('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def __init__(self):
      print ("Calling child constructor")

   def childMethod(self):
      print ('Calling child method')

c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result-

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

In a similar way, you can drive a class from multiple parent classes as follows-

```
class A:        # define your class A
.....
class B:         # define your calss B
.....
class C(A, B):   # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationship of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns True, if the given subclass **sub** is indeed a subclass of the superclass **sup**.

- The **isinstance(obj, Class)** boolean function returns True, if *obj* is an instance of class *Class* or is an instance of a subclass of Class.

Overriding Methods

You can always override your parent class methods. One reason for overriding parent's methods is that you may want special or different functionality in your subclass.

Example

```
#!/usr/bin/python3

class Parent:        # define parent class
   def myMethod(self):
      print ('Calling parent method')

class Child(Parent): # define child class
   def myMethod(self):
      print ('Calling child method')

c = Child()          # instance of child
c.myMethod()         # child calls overridden method
```

When the above code is executed, it produces the following result-

```
Calling child method
```

Base Overloading Methods

The following table lists some generic functionality that you can override in your own classes-

SN	Method, Description & Sample Call
1	**__init__ (self [,args...])** Constructor (with any optional arguments) Sample Call : *obj = className(args)*

2	**__del__(self)** Destructor, deletes an object Sample Call : *del obj*
3	**__repr__(self)** Evaluatable string representation Sample Call : *repr(obj)*
4	**__str__(self)** Printable string representation Sample Call : *str(obj)*
5	**__cmp__ (self, x)** Object comparison Sample Call : *cmp(obj, x)*

Overloading Operators

Suppose you have created a Vector class to represent two-dimensional vectors. What happens when you use the plus operator to add them? Most likely Python will yell at you.

You could, however, define the *__add__* method in your class to perform vector addition and then the plus operator would behave as per expectation –

Example

```
#!/usr/bin/python3
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b
   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

When the above code is executed, it produces the following result-

```
Vector(7,8)
```

Data Hiding

An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then will not be directly visible to outsiders.

Example

```
#!/usr/bin/python3

class JustCounter:
   __secretCount = 0

   def count(self):
      self.__secretCount += 1
      print (self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print (counter.__secretCount)
```

When the above code is executed, it produces the following result-

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print counter.__secretCount
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python protects those members by internally changing the name to include the class name. You can access such attributes as *object._className__attrName*. If you would replace your last line as following, then it works for you-

```
.........................
print (counter._JustCounter__secretCount)
```

When the above code is executed, it produces the following result-

```
1
2
2
```

20. Python 3 – Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module **re** provides full support for Perl-like regular expressions in Python. The **re** module raises the exception **re.error** if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. Nevertheless, a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as**r'expression'**.

Basic patterns that match single chars

- **a, X, 9, <** -- ordinary characters just match themselves exactly.
- **. (a period)** -- matches any single character except newline '\n'
- **\w** -- matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_].
- **\W** -- matches any non-word character.
- **\b** -- boundary between word and non-word
- **\s** -- matches a single whitespace character -- space, newline, return, tab
- **\S** -- matches any non-whitespace character.
- **\t, \n, \r** -- tab, newline, return
- **\d** -- decimal digit [0-9]
- **^** = matches start of the string
- **$** = match the end of the string
- **** -- inhibit the "specialness" of a character.

Compilation flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the **re** module under two names, a long name such as **IGNORECASE** and a short, one-letter form such as I.

Flag	Meaning
ASCII, A	Makes several escapes like \w, \b, \s and \d match only on ASCII characters with the respective property.
DOTALL, S	Make, match any character, including newlines
IGNORECASE, I	Do case-insensitive matches

LOCALE, L	Do a locale-aware match
MULTILINE, M	Multi-line matching, affecting ^ and $
VERBOSE, X (for 'extended')	Enable verbose REs, which can be organized more cleanly and understandably

The match Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function-

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters-

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern at the beginning of string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)
groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python3
import re


line = "Cats are smarter than dogs"
```

```
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
if matchObj:
   print ("matchObj.group() : ", matchObj.group())
   print ("matchObj.group(1) : ", matchObj.group(1))
   print ("matchObj.group(2) : ", matchObj.group(2))
else:
   print ("No match!!")
```

When the above code is executed, it produces the following result-

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The search Function

This function searches for first occurrence of RE *pattern* within the *string,* with optional *flags*.

Here is the syntax for this function-

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters-

Parameter	Description
pattern	This is the regular expression to be matched.
string	This is the string, which would be searched to match the pattern anywhere in the string.
flags	You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get the matched expression.

Match Object Methods	Description
group(num=0)	This method returns entire match (or specific subgroup num)

groups()	This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python3
import re

line = "Cats are smarter than dogs";
searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)
if searchObj:
   print ("searchObj.group() : ", searchObj.group())
   print ("searchObj.group(1) : ", searchObj.group(1))
   print ("searchObj.group(2) : ", searchObj.group(2))
else:
   print ("Nothing found!!")
```

When the above code is executed, it produces following result-

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

Matching Versus Searching

Python offers two different primitive operations based on regular expressions :**match** checks for a match only at the beginning of the string, while **search** checks for a match anywhere in the string (this is what Perl does by default).

Example

```
#!/usr/bin/python3
import re
line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print ("match --> matchObj.group() : ", matchObj.group())
else:
   print ("No match!!")
searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
```

```
    print ("search --> searchObj.group() : ", searchObj.group())
else:
    print ("Nothing found!!")
```

When the above code is executed, it produces the following result-

```
No match!!
search --> matchObj.group() :  dogs
```

Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* is provided. This method returns modified string.

Example

```
#!/usr/bin/python3
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print ("Phone Num : ", num)

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print ("Phone Num : ", num)
```

When the above code is executed, it produces the following result-

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple

modifiers using exclusive OR (|), as shown previously and may be represented by one of these-

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

Except for the control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

The following table lists the regular expression syntax that is available in Python-

Pattern	Description
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets

re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?imx)	Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?-imx)	Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
(?: re)	Groups regular expressions without remembering matched text.
(?imx: re)	Temporarily toggles on i, m, or x options within parentheses.
(?-imx: re)	Temporarily toggles off i, m, or x options within parentheses.
(?#...)	Comment.
(?= re)	Specifies position using a pattern. Does not have a range.
(?! re)	Specifies position using pattern negation. Does not have a range.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.

\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\1...\9	Matches nth grouped subexpression.
\10	Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular Expression Examples

Literal characters

Example	Description
python	Match "python".

Character classes

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

Special Character Classes

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a nondigit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match nonwhitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Example	Description
ruby?	Match "rub" or "ruby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\d{3}	Match exactly 3 digits
\d{3,}	Match 3 or more digits
\d{3,5}	Match 3, 4, or 5 digits

Nongreedy Repetition

This matches the smallest number of repetitions-

Example	Description
<.*>	Greedy repetition: matches "<python>perl>"
<.*?>	Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Example	Description
\D\d+	No group: + repeats \d
(\D\d)+	Grouped: + repeats \D\d pair
([Pp]ython(,)?)+	Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again-

Example	Description
([Pp])ython&\1ails	Match python&pails or Python&Pails
(['"])[^\1]*\1	Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

<table>
<tr><th>Example</th><th>Description</th></tr>
<tr><td>python|perl</td><td>Match "python" or "perl"</td></tr>
<tr><td>rub(y|le))</td><td>Match "ruby" or "ruble"</td></tr>
<tr><td>Python(!+|\?)</td><td>"Python" followed by one or more ! or one ?</td></tr>
</table>

Anchors

This needs to specify match position.

Example	Description
^Python	Match "Python" at the start of a string or internal line
Python$	Match "Python" at the end of a string or line
\APython	Match "Python" at the start of a string
Python\Z	Match "Python" at the end of a string
\bPython\b	Match "Python" at a word boundary
\brub\B	\B is nonword boundary: match "rub" in "rube" and "ruby" but not alone

Python(?=!)	Match "Python", if followed by an exclamation point.
Python(?!!)	Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Example	Description
R(?#comment)	Matches "R". All the rest is a comment
R(?i)uby	Case-insensitive while matching "uby"
R(?i:uby)	Same as above
rub(?:y|le))	Group only without creating \1 backreference

21. Python 3 – CGI Programming

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA and NCSA.

What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.

- The current version is CGI/1.1 and CGI/1.2 is under progress.

Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyperlink to browse a particular web page or URL.

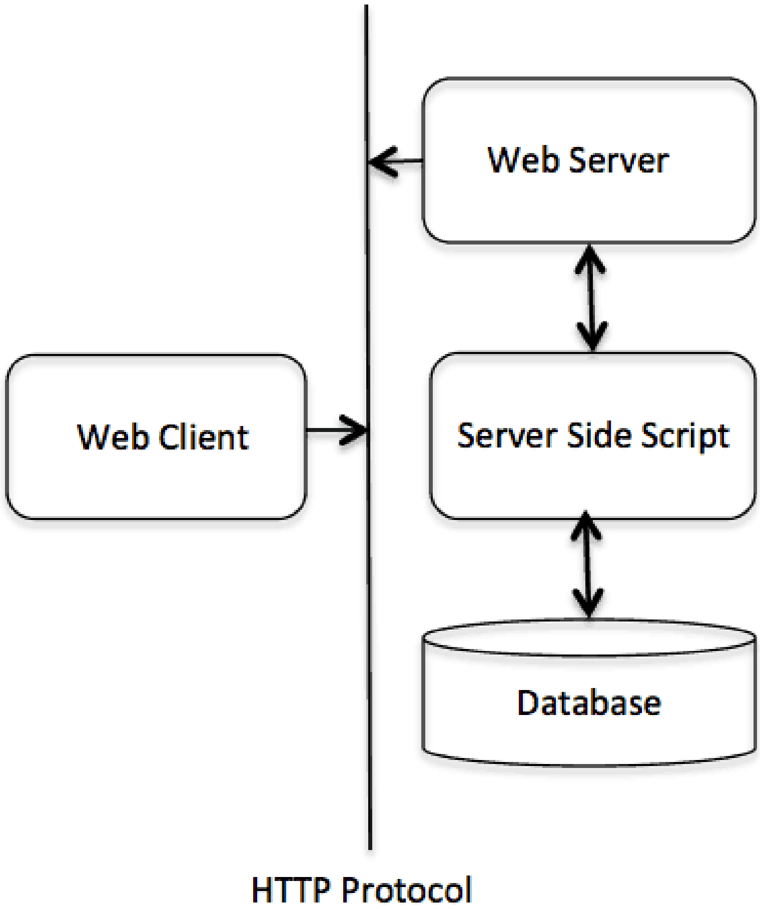
- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.

- The web server parses the URL and looks for the filename. If it finds the particular file, then it sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.

- The web browser takes response from the web server and displays either, the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back. Instead, it is executed as a program, and whatever that output of the program, is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be Python Script, PERL Script, Shell Script, C or C++ program, etc.

CGI Architecture Diagram



Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs, which are to be executed by the HTTP server, are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as /var/www/cgi-bin. By convention, CGI files have extension as **.cgi,** but you can keep your files with python extension **.py** as well.

By default, the Linux server is configured to run only the scripts in the cgi-bin directory in /var/www. If you want to specify any other directory to run your CGI scripts, comment the following lines in the httpd.conf file –

```
<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>
```

```
<Directory "/var/www/cgi-bin">
Options All
</Directory>
```

The following line should also be added for apache server to treat .py file as cgi script.

```
AddHandler cgi-script .py
```

Here, we assume that you have Web Server up and running successfully and you are able to run any other CGI program like Perl or Shell, etc.

First CGI Program

Here is a simple link, which is linked to a CGI script called hello.py. This file is kept in /var/www/cgi-bin directory and it has the following content. Before running your CGI program, make sure you have changed the mode of file using **chmod 755 hello.py,** the UNIX command to make file executable.

```
#!/usr/bin/python3

print ("Content-type:text/html")
print()
print ("<html>")
print ('<head>')
print ('<title>Hello Word - First CGI Program</title>')
print ('</head>')
print ('<body>')
print ('<h2>Hello Word! This is my first CGI program</h2>')
print ('</body>')
print ('</html>')
```

Note: First line in the script must be the path to Python executable. In Linux, it should be #!/usr/bin/python3

Enter the following URL in your browser -

```
http://www.tutorialspoint.com/cgi-bin/hello.py
```

Hello Word! This is my first CGI program

This hello.py script is a simple Python script, which writes its output on STDOUT file, i.e., the screen. There is one important and extra feature available that is the first line to be printed **Content-type:text/html** followed by a blank line. This line is sent back to the browser and it specifies the content type to be displayed on the browser screen.

By now, you must have understood the basic concept of CGI and you can write many complicated CGI programs using Python. This script can interact with any other external system also to exchange information such as RDBMS.

HTTP Header

The line **Content-type:text/html\r\n\r\n** is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form-

```
HTTP Field Name: Field Content
For Example
Content-type: text/html\r\n\r\n
```

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

Header	Description
Content-type:	A MIME string defining the format of the file being returned. Example is Content-type:text/html
Expires: Date	The date the information becomes invalid. It is used by the browser to decide when a page needs to be refreshed. A valid date string is in the format 01 Jan 1998 12:00:00 GMT.
Location: URL	The URL that is returned instead of the URL requested. You can use this field to redirect a request to any file.
Last-modified: Date	The date of last modification of the resource.
Content-length: N	The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.
Set-Cookie: String	Set the cookie passed through the *string*

CGI Environment Variables

All the CGI programs have access to the following environment variables. These variables play an important role while writing any CGI program.

Variable Name	Description

CONTENT_TYPE	The data type of the content. Used when the client is sending attached content to the server. For example, file upload.
CONTENT_LENGTH	The length of the query information. It is available only for POST requests.
HTTP_COOKIE	Returns the set cookies in the form of key & value pair.
HTTP_USER_AGENT	The User-Agent request-header field contains information about the user agent originating the request. It is name of the web browser.
PATH_INFO	The path for the CGI script.
QUERY_STRING	The URL-encoded information that is sent with GET method request.
REMOTE_ADDR	The IP address of the remote host making the request. This is useful logging or for authentication.
REMOTE_HOST	The fully qualified name of the host making the request. If this information is not available, then REMOTE_ADDR can be used to get IR address.
REQUEST_METHOD	The method used to make the request. The most common methods are GET and POST.
SCRIPT_FILENAME	The full path to the CGI script.
SCRIPT_NAME	The name of the CGI script.
SERVER_NAME	The server's hostname or IP Address
SERVER_SOFTWARE	The name and version of the software the server is running.

Here is a small CGI program to list out all the CGI variables. Click this link to see the result Get Environment.

```
#!/usr/bin/python3
import os
print ("Content-type: text/html")
print ()
print ("<font size=+1>Environment</font><\br>";)
```

```
for param in os.environ.keys():
  print ("<b>%20s</b>: %s<\br>" % (param, os.environ[param]))
```

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, a browser uses two methods to pass this information to the web server. These methods are GET Method and POST Method.

Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows-

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

- The GET method is the default method to pass information from the browser to the web server and it produces a long string that appears in your browser's Location:box.
- Never use GET method if you have password or other sensitive information to pass to the server.
- The GET method has size limtation: only 1024 characters can be sent in a request string.
- The GET method sends information using QUERY_STRING header and will be accessible in your CGI Program through QUERY_STRING environment variable.

You can pass information by simply concatenating key and value pairs along with any URL or you can use HTML <FORM> tags to pass information using GET method.

Simple URL Example – Get Method

Here is a simple URL, which passes two values to hello_get.py program using GET method.

```
/cgi-bin/hello_get.py?first_name=Malhar&last_name=Lathkar
```

Given below is the **hello_get.py** script to handle the input given by web browser. We are going to use the **cgi** module, which makes it very easy to access the passed information-

```
#!/usr/bin/python3

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()
```

```
# Get data from fields
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print ("Content-type:text/html")
print()
print ("<html>)"
print ("<head>")
print ("<title>Hello - Second CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2>Hello %s %s</h2>" % (first_name, last_name))
print ("</body>")
print ("</html>">)
```

This would generate the following result-

Hello ZARA ALI

Simple FORM Example – GET Method

This example passes two values using HTML FORM and submit button. We use the same CGI script hello_get.py to handle this input.

```
<form action="/cgi-bin/hello_get.py" method="get">
First Name: <input type="text" name="first_name">  <br />

Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
```

Here is the actual output of the above form, you enter the First and the Last Name and then click submit button to see the result.

First Name:

Last Name: Submit

Passing Information Using POST Method

A generally more reliable method of passing information to a CGI program is the POST method. This packages the information in exactly the same way as the GET methods, but instead of sending it as a text string after a ? in the URL, it sends it as a separate message. This message comes into the CGI script in the form of the standard input.

Given below is same hello_get.py script, which handles GET as well as the POST method.

```
#!/usr/bin/python3

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')

print ("Content-type:text/html")
print()
print ("<html>")
print ("<head>")
print ("<title>Hello - Second CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2>Hello %s %s</h2>" % (first_name, last_name))
print ("</body>")
print ("</html>")
```

Let us again take the same example as above, which passes two values using the HTML FORM and the submit button. We use the same CGI script hello_get.py to handle this input.

```
<form action="/cgi-bin/hello_get.py" method="post">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />

<input type="submit" value="Submit" />
</form>
```

Here is the actual output of the above form. You enter the First and the Last Name and then click the submit button to see the result.



Passing Checkbox Data to CGI Program

Checkboxes are used when more than one option is required to be selected.

Here is an HTML code example for a form with two checkboxes-

```
<form action="/cgi-bin/checkbox.py" method="POST" target="_blank">
<input type="checkbox" name="maths" value="on" /> Maths
<input type="checkbox" name="physics" value="on" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

The result of this code is in the above form-



Given below is the checkbox.cgi script to handle the input given by web browser for checkbox button.

```
#!/usr/bin/python3

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('maths'):
   math_flag = "ON"
else:
   math_flag = "OFF"

if form.getvalue('physics'):
   physics_flag = "ON"
else:
```

```
    physics_flag = "OFF"

print ("Content-type:text/html")
print()
print ("<html>")
print ("<head>")
print ("<title>Checkbox - Third CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2> CheckBox Maths is : %s</h2>" % math_flag)
print ("<h2> CheckBox Physics is : %s</h2>" % physics_flag)
print ("</body>")
print ("</html>")
```

Passing Radio Button Data to CGI Program

Radio Buttons are used when only one option is required to be selected.

Here is an HTML code example for a form with two radio buttons-

```
<form action="/cgi-bin/radiobutton.py" method="post" target="_blank">
<input type="radio" name="subject" value="maths" /> Maths
<input type="radio" name="subject" value="physics" /> Physics
<input type="submit" value="Select Subject" />
</form>
```

The result of this code is the following form-



Below is radiobutton.py script to handle input given by web browser for radio button-

```
 #!/usr/bin/python3

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('subject'):
```

```
    subject = form.getvalue('subject')
else:
    subject = "Not set"

print "Content-type:text/html")
print()
print ("<html>")
print ("<head>")
print ("<title>Radio - Fourth CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2> Selected Subject is %s</h2>" % subject)
print ("</body>")
print ("</html>")
```

Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is an HTML code example for a form with a TEXTAREA box-

```
<form action="/cgi-bin/textarea.py" method="post" target="_blank">
<textarea name="textcontent" cols="40" rows="4">
Type your text here...
</textarea>
<input type="submit" value="Submit" />
</form>
```

The result of this code is the following form-



Given below is the textarea.cgi script to handle input given by web browser-

```
#!/usr/bin/python3

# Import modules for CGI handling
import cgi, cgitb

# Create instance of FieldStorage
```

```
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('textcontent'):
   text_content = form.getvalue('textcontent')
else:
   text_content = "Not entered"

print "Content-type:text/html")
print()
print ("<html>")
print ("<head>";)
print ("<title>Text Area - Fifth CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2> Entered Text Content is %s</h2>" % text_content)
print ("</body>")
```

Passing Drop Down Box Data to CGI Program

The Drop-Down Box is used when we have many options available but only one or two are selected.

Here is an HTML code example for a form with one drop-down box-

```
<form action="/cgi-bin/dropdown.py" method="post" target="_blank">
<select name="dropdown">
<option value="Maths" selected>Maths</option>
<option value="Physics">Physics</option>
</select>
<input type="submit" value="Submit"/>
</form>
```

The result of this code is the following form-



Following is the dropdown.py script to handle the input given by web browser.

```
#!/usr/bin/python3
# Import modules for CGI handling
import cgi, cgitb
```

```
# Create instance of FieldStorage
form = cgi.FieldStorage()

# Get data from fields
if form.getvalue('dropdown'):
   subject = form.getvalue('dropdown')
else:
   subject = "Not entered"

print "Content-type:text/html")
print()
print ("<html>")
print ("<head>")
print ("<title>Dropdown Box - Sixth CGI Program</title>")
print ("</head>")
print ("<body>")
print ("<h2> Selected Subject is %s</h2>" % subject)
print ("</body>")
print ("</html>")
```

Using Cookies in CGI

HTTP protocol is a stateless protocol. For a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. How to maintain user's session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of five variable-length fields-

- **Expires:** The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.

- **Domain:** The domain name of your site.
- **Path:** The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure:** If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name=Value:** Cookies are set and retrieved in the form of key and value pairs.

Setting up Cookies

It is very easy to send cookies to the browser. These cookies are sent along with the HTTP Header before the Content-type field is sent. Assuming you want to set the User ID and Password as cookies, Cookies are set as follows-

```
#!/usr/bin/python3

print ("Set-Cookie:UserID=XYZ;\r\n")
print ("Set-Cookie:Password=XYZ123;\r\n")
print ("Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n")
print ("Set-Cookie:Domain=www.tutorialspoint.com;\r\n")
print ("Set-Cookie:Path=/perl;\n")
print ("Content-type:text/html\r\n\r\n")
..........Rest of the HTML Content....
```

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set the cookies.

It is optional to set cookies attributes like Expires, Domain, and Path. It is notable that the cookies are set before sending the magic line **"Content-type:text/html\r\n\r\n**.

Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP_COOKIE and they will have the following form-

```
key1=value1;key2=value2;key3=value3....
```

Here is an example of how to retrieve cookies-

```
#!/usr/bin/python3

# Import modules for CGI handling
from os import environ
import cgi, cgitb
```

```
if environ.has_key('HTTP_COOKIE'):
   for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
      (key, value ) = split(cookie, '=');
      if key == "UserID":
         user_id = value

      if key == "Password":
         password = value
print ("User ID  = %s" % user_id)
print ("Password = %s" % password)
```

This produces the following result for the cookies set by the above script-

```
User ID = XYZ
Password = XYZ123
```

File Upload Example

To upload a file, the HTML form must have the enctype attribute set to **multipart/form-data**. The input tag with the file type creates a "Browse" button.

```
<html>
<body>
   <form enctype="multipart/form-data"
                     action="save_file.py" method="post">
   <p>File: <input type="file" name="filename" /></p>
   <p><input type="submit" value="Upload" /></p>
   </form>
</body>
</html>
```

The result of this code is the following form-

File:

Upload

The above example has been disabled intentionally to save the people from uploading the file on our server, but you can try the above code with your server.

Here is the script **save_file.py** to handle file upload-

```
#!/usr/bin/python3

import cgi, os
import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# Get filename here.
fileitem = form['filename']

# Test if the file was uploaded
if fileitem.filename:
   # strip leading path from file name to avoid
   # directory traversal attacks
   fn = os.path.basename(fileitem.filename)
   open('/tmp/' + fn, 'wb').write(fileitem.file.read())

   message = 'The file "' + fn + '" was uploaded successfully'

else:
   message = 'No file was uploaded'

print ("""\
Content-Type: text/html\n
<html>
<body>
   <p>%s</p>
</body>
</html>
""" % (message,))
```

If you run the above script on Unix/Linux, then you need to take care of replacing file separator as follows, otherwise on your windows machine above open() statement should work fine.

```
fn = os.path.basename(fileitem.filename.replace("\\", "/" ))
```

How To Raise a "File Download" Dialog Box ?

Sometimes, it is desired that you want to give an option where a user can click a link and it will pop up a "File Download" dialogue box to the user instead of displaying actual content. This is very easy and can be achieved through HTTP header. This HTTP header is different from the header mentioned in the previous section.

For example, if you want make a **FileName** file downloadable from a given link, then its syntax is as follows-

```
#!/usr/bin/python3


# HTTP Header
print ("Content-Type:application/octet-stream; name=\"FileName\"\r\n")
print ("Content-Disposition: attachment; filename=\"FileName\"\r\n\n")


# Actual File Content will go hear.
fo = open("foo.txt", "rb")


str = fo.read()
print (str)


# Close opened file
fo.close()
```

22. Python 3 – MySQL Database Access

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

You can choose the right database for your application. Python Database API supports a wide range of database servers such as –

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase
- SQLite

Here is the list of available Python database interfaces: Python Database Interfaces and APIs. You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following:

- Importing the API module.
- Acquiring a connection with the database.
- Issuing SQL statements and stored procedures.
- Closing the connection

Python has an in-built support for SQLite. In this section, we would learn all the concepts using MySQL. MySQLdb module, a popular interface with MySQL is not compatible with Python 3. Instead, we shall use PyMySQL module.

What is PyMySQL ?

PyMySQL is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library. The goal of PyMySQL is to be a drop-in replacement for MySQLdb .

How do I Install PyMySQL?

Before proceeding further, you make sure you have PyMySQL installed on your machine. Just type the following in your Python script and execute it-

```
#!/usr/bin/python3


import PyMySQL
```

If it produces the following result, then it means MySQLdb module is not installed-

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    Import PyMySQL
ImportError: No module named PyMySQL
```

The last stable release is available on PyPI and can be installed with pip:

```
pip install PyMySQL
```

Alternatively (e.g. if pip is not available), a tarball can be downloaded from GitHub and installed with Setuptools as follows-

```
$ # X.X is the desired PyMySQL version (e.g. 0.5 or 0.6).
$ curl -L https://github.com/PyMySQL/PyMySQL/tarball/pymysql-X.X | tar xz
$ cd PyMySQL*
$ python setup.py install
$ # The folder PyMySQL* can be safely removed now.
```

Note: Make sure you have root privilege to install the above module.

Database Connection

Before connecting to a MySQL database, make sure of the following points-

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Python module PyMySQL is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Example

Following is an example of connecting with MySQL database "TESTDB"-

```
#!/usr/bin/python3
import PyMySQL
# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# execute SQL query using execute() method.
cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()

print ("Database version : %s " % data)

# disconnect from server
db.close()
```

While running this script, it produces the following result-

```
Database version : 5.5.20-log
```

If a connection is established with the datasource, then a Connection Object is returned and saved into **db** for further use, otherwise **db** is set to None. Next, **db** object is used to create a **cursor** object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that the database connection is closed and resources are released.

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using **execute** method of the created cursor.

Example

Let us create a Database table EMPLOYEE-

```
#!/usr/bin/python3

import PyMySQL
```

```
# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)

# disconnect from server
db.close()
```

INSERT Operation

The INSERT Operation is required when you want to create your records into a database table.

Example

The following example, executes SQL *INSERT* statement to create a record in the EMPLOYEE table-

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
```

```
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

The above example can be written as follows to create SQL queries dynamically-

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
```

```
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Example

The following code segment is another form of execution where you can pass parameters directly-

```
..................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
             (user_id, password))
..................................
```

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once the database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch a single record or **fetchall()** method to fetch multiple values from a database table.

- **fetchone():** It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **fetchall():** It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.
- **rowcount:** This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000-

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to INSERT a record into the database.
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # Now print fetched result
      print ("fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
             (fname, lname, age, sex, income ))
except:
   print ("Error: unable to fecth data")

# disconnect from server
db.close()
```

This will produce the following result-

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as **'M'**. Here, we increase the AGE of all the males by one year.

Example

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to UPDATE required records
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = '%c'" % ('M')
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20-

Example

```
#!/usr/bin/python3

import PyMySQL

# Open database connection
db = PyMySQL.connect("localhost","testuser","test123","TESTDB" )

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Performing Transactions

Transactions are a mechanism that ensure data consistency. Transactions have the following four properties-

- **Atomicity:** Either a transaction completes or nothing happens at all.
- **Consistency:** A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation:** Intermediate results of a transaction are not visible outside the current transaction.

- **Durability:** Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

Example

You already know how to implement transactions. Here is a similar example-

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

COMMIT Operation

Commit is an operation, which gives a green signal to the database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call the **commit** method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use the **rollback()** method.

Here is a simple example to call the **rollback()** method.

```
db.rollback()
```

Disconnecting Database

To disconnect the Database connection, use the close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any

of the DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already cancelled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Exception	Description
Warning	Used for non-fatal issues. Must subclass StandardError.
Error	Base class for errors. Must subclass StandardError.
InterfaceError	Used for errors in the database module, not the database itself. Must subclass Error.
DatabaseError	Used for errors in the database. Must subclass Error.
DataError	Subclass of DatabaseError that refers to errors in the data.
OperationalError	Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
IntegrityError	Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
InternalError	Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
ProgrammingError	Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
NotSupportedError	Subclass of DatabaseError that refers to trying to call unsupported functionality.

Your Python scripts should handle these errors, but before using any of the above exceptions, make sure your MySQLdb has support for that exception. You can get more information about them by reading the DB API 2.0 specification.

23. Python 3 – Network Programming

Python provides two levels of access to the network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This chapter gives you an understanding on the most famous concept in Networking - Socket Programming.

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary-

Term	Description
domain	The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
type	The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
protocol	Typically zero, this may be used to identify a variant of a protocol within a domain and type.
hostname	The identifier of a network interface: A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation A string "<broadcast>", which specifies an INADDR_BROADCAST address. A zero-length string, which specifies INADDR_ANY, or

	An Integer, interpreted as a binary address in host byte order.
port	Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

The socket Module

To create a socket, you must use the socket.socket() function available in the socket module, which has the general syntax-

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

Here is the description of the parameters-

- **socket_family:** This is either AF_UNIX or AF_INET, as explained earlier.
- **socket_type:** This is either SOCK_STREAM or SOCK_DGRAM.
- **protocol:** This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use the required functions to create your client or server program. Following is the list of functions required-

Server Socket Methods

Method	Description
s.bind()	This method binds address (hostname, port number pair) to socket.
s.listen()	This method sets up and start TCP listener.
s.accept()	This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods

Method	Description
s.connect()	This method actively initiates TCP server connection.

General Socket Methods

Method	Description
s.recv()	This method receives TCP message
s.send()	This method transmits TCP message
s.recvfrom()	This method receives UDP message
s.sendto()	This method transmits UDP message
s.close()	This method closes socket
socket.gethostname()	Returns the hostname.

A Simple Server

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call the **bind(hostname, port)** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python3           # This is server.py file
import socket

# create a socket object
serversocket = socket.socket(
              socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()

port = 9999
```

```
# bind to the port
serversocket.bind((host, port))

# queue up to 5 requests
serversocket.listen(5)

while True:
    # establish a connection
    clientsocket,addr = serversocket.accept()

    print("Got a connection from %s" % str(addr))

    msg='Thank you for connecting'+ "\r\n"
    clientsocket.send(msg.encode('ascii'))
    clientsocket.close()
```

A Simple Client

Let us write a very simple client program, which opens a connection to a given port 12345 and a given host. It is very simple to create a socket client using the Python's *socket* module function.

The **socket.connect(hosname, port)** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits-

```
#!/usr/bin/python3           # This is client.py file
import socket
# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# get local machine name
host = socket.gethostname()
port = 9999
# connection to hostname on the port.
s.connect((host, port))
# Receive no more than 1024 bytes
msg = s.recv(1024)
s.close()
```

```
print (msg.decode('ascii'))
```

Now run this server.py in the background and then run the above client.py to see the result.

```
# Following would start a server in background.
$ python server.py &
# Once server is started run client as follows:
$ python client.py
```

This would produce the following result-

```
on server terminal


Got a connection from ('192.168.1.10', 3747)
On client terminal


Thank you for connecting
```

Python Internet Modules

A list of some important modules in Python Network/Internet programming are given below.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols.

Further Readings

This was a quick start with the Socket Programming. It is a vast subject. It is recommended to go through the following link to find more detail-

- Unix Socket Programming.
- Python Socket Library and Modules.

24. Python 3 – Sending Email using SMTP

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending an e-mail and routing e-mail between mail servers.

Python provides smtplib module, which defines an SMTP client session object that can be used to send mails to any Internet machine with an SMTP or ESMTP listener daemon.

Here is a simple syntax to create one SMTP object, which can later be used to send an e-mail-

```
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

Here is the detail of the parameters-

- **host:** This is the host running your SMTP server. You can specifiy IP address of the host or a domain name like tutorialspoint.com. This is an optional argument.

- **port:** If you are providing *host* argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.

- **local_hostname**: If your SMTP server is running on your local machine, then you can specify just *localhost* as the option.

An SMTP object has an instance method called **sendmail**, which is typically, used to do the work of mailing a message. It takes three parameters-

- The *sender* - A string with the address of the sender.
- The *receivers* - A list of strings, one for each recipient.
- The *message* - A message as a string formatted as specified in the various RFCs.

Example

Here is a simple way to send one e-mail using Python script. Try it once-

```
#!/usr/bin/python3

import smtplib

sender ='from@fromdomain.com'
receivers =['to@todomain.com']

message ="""From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test
```

```
This is a test e-mail message.
"""
try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
print ("Successfully sent email")
except smtplib.SMTPException:
print ("Error: unable to send email")
```

Here, you have placed a basic e-mail in message, using a triple quote, taking care to format the headers correctly. An e-mail requires a **From**, **To**, and a **Subject** header, separated from the body of the e-mail with a blank line.

To send the mail you use *smtpObj* to connect to the SMTP server on the local machine. Then use the *sendmail* method along with the message, the from address, and the destination address as parameters (even though the from and to addresses are within the e-mail itself, these are not always used to route the mail).

If you are not running an SMTP server on your local machine, you can the use*smtplib* client to communicate with a remote SMTP server. Unless you are using a webmail service (such as gmail or Yahoo! Mail), your e-mail provider must have provided you with the outgoing mail server details that you can supply them, as follows-

```
mail=smtplib.SMTP('smtp.gmail.com', 587)
```

Sending an HTML e-mail using Python

When you send a text message using Python, then all the content is treated as simple text. Even if you include HTML tags in a text message, it is displayed as simple text and HTML tags will not be formatted according to the HTML syntax. However, Python provides an option to send an HTML message as actual HTML message.

While sending an e-mail message, you can specify a Mime version, content type and the character set to send an HTML e-mail.

Example

Following is an example to send the HTML content as an e-mail. Try it once-

```
#!/usr/bin/python3

import smtplib

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
```

```
MIME-Version: 1.0
Content-type: text/html
Subject: SMTP HTML e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>
<h1>This is headline.</h1>
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

Sending Attachments as an E-mail

To send an e-mail with mixed content requires setting the **Content-type** header to **multipart/mixed**. Then, the text and the attachment sections can be specified within **boundaries**.

A boundary is started with two hyphens followed by a unique number, which cannot appear in the message part of the e-mail. A final boundary denoting the e-mail's final section must also end with two hyphens.

The attached files should be encoded with the **pack("m")** function to have base 64 encoding before transmission.

Example

Following is an example, which sends a file /tmp/test.txt as an attachment. Try it once-

```
#!/usr/bin/python3

import smtplib
import base64

filename = "/tmp/test.txt"

# Read a file and encode it into base64 format
fo = open(filename, "rb")
filecontent = fo.read()
```

```
encodedcontent = base64.b64encode(filecontent)  # base64
sender = 'webmaster@tutorialpoint.com'
reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"
body ="""
This is a test email to send an attachement.
"""
# Define the main headers.
part1 = """From: From Person <me@fromdomain.net>
To: To Person <amrood.admin@gmail.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=%s
--%s
""" % (marker, marker)

# Define the message action
part2 = """Content-Type: text/plain
Content-Transfer-Encoding:8bit

%s
--%s
""" % (body,marker)

# Define the attachment section
part3 = """Content-Type: multipart/mixed; name=\"%s\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename=%s

%s
--%s--
""" %(filename, filename, encodedcontent, marker)
message = part1 + part2 + part3

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, reciever, message)
```

```
    print ("Successfully sent email")
except Exception:
    print ("Error: unable to send email")
```

25. Python 3 – Multithreaded Programming

Running several threads is similar to running several different programs concurrently, but with the following benefits-

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

- Threads are sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context is it currently running.

- It can be pre-empted (interrupted).

- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

There are two different kind of threads-

- kernel thread

- user thread

Kernel Threads are a part of the operating system, while the User-space threads are not implemented in the kernel.

There are two modules, which support the usage of threads in Python3-

- _thread

- threading

The thread module has been "deprecated" for quite a long time. Users are encouraged to use the threading module instead. Hence, in Python 3, the module "thread" is not available anymore. However, it has been renamed to "_thread" for backward compatibilities in Python3.

Starting a New Thread

To spawn another thread, you need to call the following method available in the *thread* module-

```
_thread.start_new_thread ( function, args[, kwargs] )
```

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*. When the function returns, the thread terminates.

Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

Example

```
#!/usr/bin/python3

import _thread
import time

# Define a function for the thread
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print ("%s: %s" % ( threadName, time.ctime(time.time()) ))

# Create two threads as follows
try:
   _thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   _thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print ("Error: unable to start thread")

while 1:
   pass
```

When the above code is executed, it produces the following result-

```
Thread-1: Fri Feb 19 09:41:39 2016
Thread-2: Fri Feb 19 09:41:41 2016
Thread-1: Fri Feb 19 09:41:41 2016
Thread-1: Fri Feb 19 09:41:43 2016
Thread-2: Fri Feb 19 09:41:45 2016
Thread-1: Fri Feb 19 09:41:45 2016
Thread-1: Fri Feb 19 09:41:47 2016
Thread-2: Fri Feb 19 09:41:49 2016
Thread-2: Fri Feb 19 09:41:53 2016
```

Program goes in an infinite loop. You will have to press ctrl-c to stop.

Although it is very effective for low-level threading, the *thread* module is very limited compared to the newer threading module.

The Threading Module

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

- **threading.activeCount():** Returns the number of thread objects that are active.
- **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
- **threading.enumerate():** Returns a list of all the thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

- **run():** The run() method is the entry point for a thread.
- **start():** The start() method starts a thread by calling the run method.
- **join([time]):** The join() waits for threads to terminate.
- **isAlive():** The isAlive() method checks whether a thread is still executing.
- **getName():** The getName() method returns the name of a thread.
- **setName():** The setName() method sets the name of a thread.

Creating Thread Using Threading Module

To implement a new thread using the threading module, you have to do the following –

- Define a new subclass of the *Thread* class.
- Override the *__init__(self [,args])* method to add additional arguments.
- Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls the *run()*method.

Example

```
#!/usr/bin/python3
```

```
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("Starting " + self.name)
        print_time(self.name, self.counter, 5)
        print ("Exiting " + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("Exiting Main Thread")
```

When we run the above program, it produces the following result-

```
Starting Thread-1
Starting Thread-2
Thread-1: Fri Feb 19 10:00:21 2016
Thread-2: Fri Feb 19 10:00:22 2016
Thread-1: Fri Feb 19 10:00:22 2016
```

```
Thread-1: Fri Feb 19 10:00:23 2016
Thread-2: Fri Feb 19 10:00:24 2016
Thread-1: Fri Feb 19 10:00:24 2016
Thread-1: Fri Feb 19 10:00:25 2016
Exiting Thread-1
Thread-2: Fri Feb 19 10:00:26 2016
Thread-2: Fri Feb 19 10:00:28 2016
Thread-2: Fri Feb 19 10:00:30 2016
Exiting Thread-2
Exiting Main Thread
```

Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force the threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

Example

```
#!/usr/bin/python3
import threading
import time
class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("Starting " + self.name)
        # Get lock to synchronize threads
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
```

```
        # Free lock to release next thread
        threadLock.release()
def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

# Add threads to thread list
threads.append(thread1)
threads.append(thread2)

# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exiting Main Thread")
```

When the above code is executed, it produces the following result-

```
Starting Thread-1
Starting Thread-2
Thread-1: Fri Feb 19 10:04:14 2016
Thread-1: Fri Feb 19 10:04:15 2016
Thread-1: Fri Feb 19 10:04:16 2016
Thread-2: Fri Feb 19 10:04:18 2016
Thread-2: Fri Feb 19 10:04:20 2016
Thread-2: Fri Feb 19 10:04:22 2016
Exiting Main Thread
```

Multithreaded Priority Queue

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue –

- **get():** The get() removes and returns an item from the queue.
- **put():** The put adds item to a queue.
- **qsize() :** The qsize() returns the number of items that are currently in the queue.
- **empty():** The empty( ) returns True if queue is empty; otherwise, False.
- **full():** the full() returns True if queue is full; otherwise, False.

Example

```
#!/usr/bin/python3

import queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print ("Starting " + self.name)
        process_data(self.name, self.q)
        print ("Exiting " + self.name)

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
```

```
        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# Fill the queue
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# Wait for queue to empty
while not workQueue.empty():
    pass

# Notify threads it's time to exit
exitFlag = 1

# Wait for all threads to complete
for t in threads:
    t.join()
print ("Exiting Main Thread")
```

When the above code is executed, it produces the following result-

```
Starting Thread-1
Starting Thread-2
Starting Thread-3
```

```
Thread-1 processing One
Thread-2 processing Two
Thread-3 processing Three
Thread-1 processing Four
Thread-2 processing Five
Exiting Thread-3
Exiting Thread-1
Exiting Thread-2
Exiting Main Thread
```

26. Python 3 – XML Processing

XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.

What is XML?

The Extensible Markup Language (XML) is a markup language much like HTML or SGML. This is recommended by the World Wide Web Consortium and available as an open standard.

XML is extremely useful for keeping track of small to medium amounts of data without requiring an SQL- based backbone.

XML Parser Architectures and APIs

The Python standard library provides a minimal but useful set of interfaces to work with XML.

The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

- **Simple API for XML (SAX):** Here, you register callbacks for events of interest and then let the parser proceed through the document. This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from the disk and the entire file is never stored in the memory.

- **Document Object Model (DOM) API:** This is a World Wide Web Consortium recommendation wherein the entire file is read into the memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX obviously cannot process information as fast as DOM, when working with large files. On the other hand, using DOM exclusively can really kill your resources, especially if used on many small files.

SAX is read-only, while DOM allows changes to the XML file. Since these two different APIs literally complement each other, there is no reason why you cannot use them both for large projects.

For all our XML code examples, let us use a simple XML file *movies.xml* as an input-

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
   <type>War, Thriller</type>
   <format>DVD</format>
   <year>2003</year>
   <rating>PG</rating>
   <stars>10</stars>
```

```
    <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
    <type>Anime, Science Fiction</type>
    <format>DVD</format>
    <year>1989</year>
    <rating>R</rating>
    <stars>8</stars>
    <description>A schientific fiction</description>
</movie>
    <movie title="Trigun">
    <type>Anime, Action</type>
    <format>DVD</format>
    <episodes>4</episodes>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
    <type>Comedy</type>
    <format>VHS</format>
    <rating>PG</rating>
    <stars>2</stars>
    <description>Viewable boredom</description>
</movie>
</collection>
```

Parsing XML with SAX APIs

SAX is a standard interface for event-driven XML parsing. Parsing XML with SAX generally requires you to create your own ContentHandler by subclassing xml.sax.ContentHandler.

Your *ContentHandler* handles the particular tags and attributes of your flavor(s) of XML. A ContentHandler object provides methods to handle various parsing events. Its owning parser calls ContentHandler methods as it parses the XML file.

The methods *startDocument* and *endDocument* are called at the start and the end of the XML file. The method *characters(text)* is passed the character data of the XML file via the parameter text.

The ContentHandler is called at the start and end of each element. If the parser is not in namespace mode, the methods *startElement(tag, attributes)* and*endElement(tag)* are

called; otherwise, the corresponding methods*startElementNS* and *endElementNS* are called. Here, tag is the element tag, and attributes is an Attributes object.

Here are other important methods to understand before proceeding-

The make_parser Method

The following method creates a new parser object and returns it. The parser object created will be of the first parser type, the system finds.

```
xml.sax.make_parser( [parser_list] )
```

Here is the detail of the parameters-

- **parser_list:** The optional argument consisting of a list of parsers to use, which must all implement the make_parser method.

The parse Method

The following method creates a SAX parser and uses it to parse a document.

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler])
```

Here are the details of the parameters-

- **xmlfile:** This is the name of the XML file to read from.
- **contenthandler:** This must be a ContentHandler object.
- **errorhandler:** If specified, errorhandler must be a SAX ErrorHandler object.

The parseString Method

There is one more method to create a SAX parser and to parse the specified**XML string**.

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

Here are the details of the parameters-

- **xmlstring:** This is the name of the XML string to read from.
- **contenthandler:** This must be a ContentHandler object.
- **errorhandler:** If specified, errorhandler must be a SAX ErrorHandler object.

Example

```
#!/usr/bin/python3


import xml.sax

```

```
class MovieHandler( xml.sax.ContentHandler ):
   def __init__(self):
      self.CurrentData = ""
      self.type = ""
      self.format = ""
      self.year = ""
      self.rating = ""
      self.stars = ""
      self.description = ""

   # Call when an element starts
   def startElement(self, tag, attributes):
      self.CurrentData = tag
      if tag == "movie":
         print ("*****Movie*****")
         title = attributes["title"]
         print ("Title:", title)

   # Call when an elements ends
   def endElement(self, tag):
      if self.CurrentData == "type":
         print ("Type:", self.type)
      elif self.CurrentData == "format":
         print ("Format:", self.format)
      elif self.CurrentData == "year":
         print ("Year:", self.year)
      elif self.CurrentData == "rating":
         print ("Rating:", self.rating)
      elif self.CurrentData == "stars":
         print ("Stars:", self.stars)
      elif self.CurrentData == "description":
         print ("Description:", self.description)
      self.CurrentData = ""

   # Call when a character is read
   def characters(self, content):
      if self.CurrentData == "type":
         self.type = content
      elif self.CurrentData == "format":
```

```
            self.format = content
         elif self.CurrentData == "year":
            self.year = content
         elif self.CurrentData == "rating":
            self.rating = content
         elif self.CurrentData == "stars":
            self.stars = content
         elif self.CurrentData == "description":
            self.description = content

if ( __name__ == "__main__"):

   # create an XMLReader
   parser = xml.sax.make_parser()
   # turn off namepsaces
   parser.setFeature(xml.sax.handler.feature_namespaces, 0)

   # override the default ContextHandler
   Handler = MovieHandler()
   parser.setContentHandler( Handler )

   parser.parse("movies.xml")
```

This would produce the following result-

```
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Year: 2003
Rating: PG
Stars: 10
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Year: 1989
Rating: R
Stars: 8
```

```
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Stars: 10
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Stars: 2
Description: Viewable boredom
```

For a complete detail on SAX API documentation, please refer to the standard Python SAX APIs.

Parsing XML with DOM APIs

The Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying the XML documents.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another.

Here is the easiest way to load an XML document quickly and to create a minidom object using the xml.dom module. The minidom object provides a simple parser method that quickly creates a DOM tree from the XML file.

The sample phrase calls the parse(file [,parser]) function of the minidom object to parse the XML file, designated by file into a DOM tree object.

```
#!/usr/bin/python3


from xml.dom.minidom import parse
import xml.dom.minidom


# Open XML document using minidom parser
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
   print ("Root element : %s" % collection.getAttribute("shelf"))
```

```
# Get all the movies in the collection
movies = collection.getElementsByTagName("movie")

# Print detail of each movie.
for movie in movies:
   print ("*****Movie*****")
   if movie.hasAttribute("title"):
      print ("Title: %s" % movie.getAttribute("title"))

   type = movie.getElementsByTagName('type')[0]
   print ("Type: %s" % type.childNodes[0].data)
   format = movie.getElementsByTagName('format')[0]
   print ("Format: %s" % format.childNodes[0].data)
   rating = movie.getElementsByTagName('rating')[0]
   print ("Rating: %s" % rating.childNodes[0].data)
   description = movie.getElementsByTagName('description')[0]
   print ("Description: %s" % description.childNodes[0].data)
```

This would produce the following result-

```
Root element : New Arrivals
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Rating: PG
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Rating: R
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
```

```
Rating: PG
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Description: Viewable boredom
```

For a complete detail on DOM API documentation, please refer to the standard Python DOM APIs.

27. Python 3 – GUI Programming (Tkinter)

Python provides various options for developing graphical user interfaces (GUIs). The most important features are listed below.

- **Tkinter:** Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look at this option in this chapter.
- **wxPython:** This is an open-source Python interface for wxWidgets GUI toolkit. You can find a complete tutorial on WxPython here.
- **PyQt:**This is also a Python interface for a popular cross-platform Qt GUI library. TutorialsPoint has a very good tutorial on PyQt here.
- **JPython:** JPython is a Python port for Java, which gives Python scripts seamless access to the Java class libraries on the local machinehttp://www.jython.org.

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps –

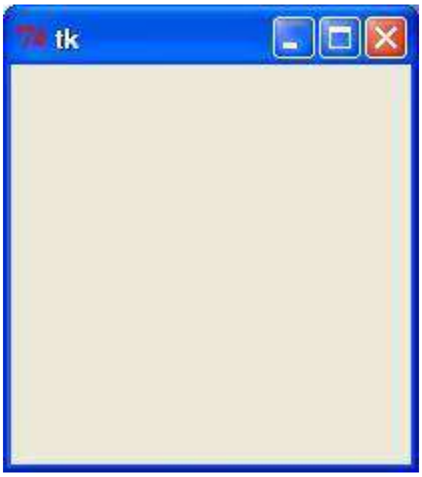
- Import the *Tkinter* module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

Example

```
#!/usr/bin/python3


import tkinter # note that module name has changed from Tkinter in Python 2 to tkinter in
Python 3
top = tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

This would create a following window-



Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table-

Operator	Description
Button	The Button widget is used to display the buttons in your application.
Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
Entry	The Entry widget is used to display a single-line text field for accepting values from a user.
Frame	The Frame widget is used as a container widget to organize other widgets.
Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.

Listbox	The Listbox widget is used to provide a list of options to a user.
Menubutton	The Menubutton widget is used to display menus in your application.
Menu	The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
Message	The Message widget is used to display multiline text fields for accepting values from a user.
Radiobutton	The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
Scale	The Scale widget is used to provide a slider widget.
Scrollbar	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
Text	The Text widget is used to display text in multiple lines.
Toplevel	The Toplevel widget is used to provide a separate window container.
Spinbox	The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
PanedWindow	A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
LabelFrame	A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
tkMessageBox	This module is used to display the message boxes in your applications.

Let us study these widgets in detail.

Tkinter Button

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

Syntax

Here is the simple syntax to create this widget-

```
w = Button ( master, option=value, ... )
```

Parameters

- **master:** This represents the parent window.

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	Background color when the button is under the cursor.
activeforeground	Foreground color when the button is under the cursor.
bd	Border width in pixels. Default is 2.
bg	Normal background color.
command	Function or method to be called when the button is clicked.
fg	Normal foreground (text) color.
font	Text font to be used for the button's label.
height	Height of the button in text lines (for textual buttons) or pixels (for images).

highlightcolor	The color of the focus highlight when the widget has focus.
image	Image to be displayed on the button (instead of text).
justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
padx	Additional padding left and right of the text.
pady	Additional padding above and below the text.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

Methods

Following are commonly used methods for this widget-

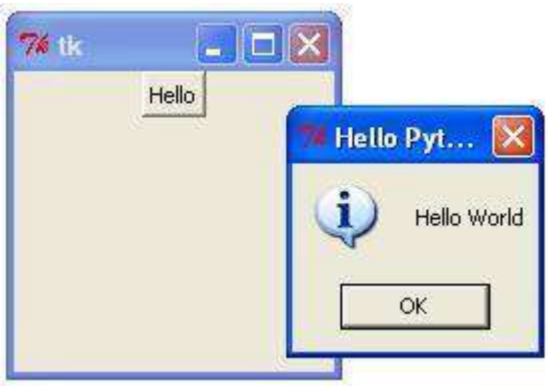
Method	Description
flash()	Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.
invoke()	Calls the button's callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *
from tkinter import messagebox
top = Tk()
top.geometry("100x100")
def helloCallBack():
   msg=messagebox.showinfo( "Hello Python", "Hello World")
B = Button(top, text ="Hello", command = helloCallBack)
B.place(x=50,y=50)
top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Canvas

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.

Syntax

Here is the simple syntax to create this widget-

```
w = Canvas ( master, option=value, ... )
```

Parameters

- **master:** This represents the parent window.

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bd	Border width in pixels. Default is 2.
bg	Normal background color.
confine	If true (the default), the canvas cannot be scrolled outside of the scrollregion.
cursor	Cursor used in the canvas like arrow, circle, dot etc.
height	Size of the canvas in the Y dimension.
highlightcolor	Color shown in the focus highlight.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
scrollregion	A tuple (w, n, e, s) that defines over how large an area the canvas can be scrolled, where w is the left side, n the top, e the right side, and s the bottom.
width	Size of the canvas in the X dimension.
xscrollincrement	If you set this option to some positive dimension, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by scrolling units, such as when the user clicks on the arrows at the ends of a scrollbar.
xscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
yscrollincrement	Works like xscrollincrement, but governs vertical movement.
yscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

The Canvas widget can support the following standard items-

arc . Creates an arc item, which can be a chord, a pieslice or a simple arc.

```
coord = 10, 50, 240, 210
```

```
arc = canvas.create_arc(coord, start=0, extent=150, fill="blue")
```

image . Creates an image item, which can be an instance of either the BitmapImage or the PhotoImage classes.

```
filename = PhotoImage(file = "sunshine.gif")
image = canvas.create_image(50, 50, anchor=NE, image=filename)
```

line . Creates a line item.

```
line = canvas.create_line(x0, y0, x1, y1, ..., xn, yn, options)
```

oval . Creates a circle or an ellipse at the given coordinates. It takes two pairs of coordinates; the top left and bottom right corners of the bounding rectangle for the oval.

```
oval = canvas.create_oval(x0, y0, x1, y1, options)
```

polygon . Creates a polygon item that must have at least three vertices.

```
oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn, options)
```

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

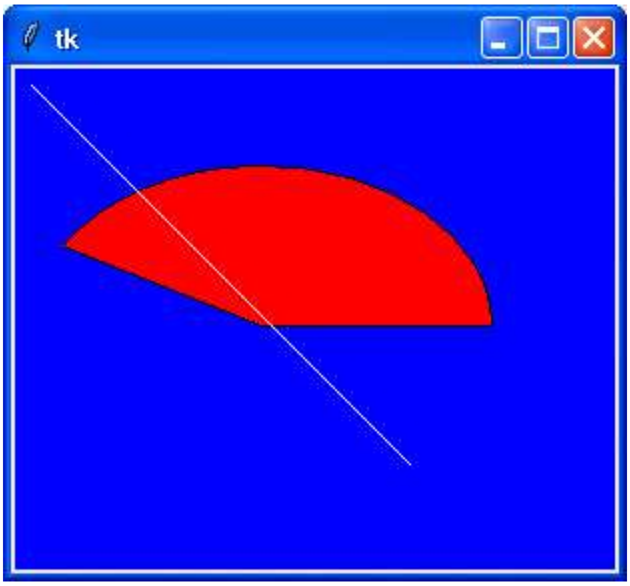
from tkinter import messagebox

top = Tk()

C = Canvas(top, bg="blue", height=250, width=300)

coord = 10, 50, 240, 210
arc = C.create_arc(coord, start=0, extent=150, fill="red")
line = C.create_line(10,10,200,200,fill='white')
C.pack()
Top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Checkbutton

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

You can also display images in place of text.

Syntax

Here is the simple syntax to create this widget-

```
w = Checkbutton ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
activebackground	Background color when the checkbutton is under the cursor.

activeforeground	Foreground color when the checkbutton is under the cursor.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a monochrome image on a button.
bd	The size of the border around the indicator. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.*
disabledforeground	The foreground color used to render the text of a disabled checkbutton. The default is a stippled version of the default foreground color.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines of text on the checkbutton. Default is 1.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
image	To display a graphic image on the button.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
offvalue	Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting offvalue to that value.
onvalue	Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting onvalue to that value.

padx	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel.
pady	How much space to leave above and below the checkbutton and text. Default is 1 pixel.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectcolor	The color of the checkbutton when it is set. Default is selectcolor="red".
selectimage	If you set this option to an image, that image will appear in the checkbutton when it is set.
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
text	The label displayed next to the checkbutton. Use newlines ("\n") to display multiple lines of text.
underline	With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
variable	The control variable that tracks the current state of the checkbutton. Normally this variable is an *IntVar*, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Methods

Following are commonly used methods for this widget-

Method	Description
deselect()	Clears (turns off) the checkbutton.
flash()	Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.
select()	Sets (turns on) the checkbutton.
toggle()	Clears the checkbutton if set, sets it if cleared.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

import tkinter

top = Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
C1 = Checkbutton(top, text = "Music", variable = CheckVar1, \
                 onvalue = 1, offvalue = 0, height=5, \
                 width = 20, )
C2 = Checkbutton(top, text = "Video", variable = CheckVar2, \
                 onvalue = 1, offvalue = 0, height=5, \
                 width = 20)
C1.pack()
C2.pack()
top.mainloop()
```

When the above code is executed, it produces the following result –



Tkinter Entry

The Entry widget is used to accept single-line text strings from a user.

- If you want to display multiple lines of text that can be edited, then you should use the *Text* widget.

- If you want to display one or more lines of text that cannot be modified by the user, then you should use the *Label* widget.

Syntax

Here is the simple syntax to create this widget-

```
w = Entry( master, option, ... )
```

Parameters

- **master:** This represents the parent window.

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.

command	A procedure to be called every time the user changes the state of this checkbutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The font used for the text.
exportselection	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
fg	The color used to render the text.
highlightcolor	The color of the focus highlight when the checkbutton has the focus.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text. The default is one pixel.
selectforeground	The foreground (text) color of selected text.
show	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
textvariable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.

width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
xscrollcommand	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Methods

Following are commonly used methods for this widget-

Method	Description
delete (first, last=None)	Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
get()	Returns the entry's current text as a string.
icursor (index)	Set the insertion cursor just before the character at the given index.
index (index)	Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
insert (index, s)	Inserts string s before the character at the given index.
select_adjust (index)	This method is used to make sure that the selection includes the character at the specified index.
select_clear()	Clears the selection. If there isn't currently a selection, has no effect.

select_from (index)	Sets the ANCHOR index position to the character selected by index, and selects that character.
select_present()	If there is a selection, returns true, else returns false.
select_range (start, end)	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
select_to (index)	Selects all the text from the ANCHOR position up to but not including the character at the given index.
xview (index)	This method is useful in linking the Entry widget to a horizontal scrollbar.
xview_scroll (number, what)	Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)

top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Frame

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.

It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax

Here is the simple syntax to create this widget-

```
w = Frame ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
height	The vertical dimension of the new frame.
highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.

highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

root = Tk()
frame = Frame(root)
frame.pack()

bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Label

This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want.

It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines.

Syntax

Here is the simple syntax to create this widget-

```
w = Label ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.

fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class StringVar, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

root = Tk()

var = StringVar()
label = Label( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Listbox

The Listbox widget is used to display a list of items from which a user can select a number of items

Syntax

Here is the simple syntax to create this widget-

```
w = Listbox ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
bg	The normal background color displayed behind the label and indicator.

bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over the listbox.
font	The font used for the text in the listbox.
fg	The color used for the text in the listbox.
height	Number of lines (not pixels!) shown in the listbox. Default is 10.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
highlightthickness	Thickness of the focus highlight.
relief	Selects three-dimensional border shading effects. The default is SUNKEN.
selectbackground	The background color to use displaying selected text.
selectmode	Determines how many items can be selected, and how mouse drags affect the selection: • **BROWSE:** Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default. • **SINGLE:** You can only select one line, and you can't drag the mouse.wherever you click button 1, that line is selected. • **MULTIPLE:** You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. • **EXTENDED:** You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
width	The width of the widget in characters. The default is 20.

xscrollcommand	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
yscrollcommand	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.

Methods

Methods on listbox objects include-

Options	Description
activate (index)	Selects the line specifies by the given index.
curselection()	Returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
delete (first, last=None)	Deletes the lines whose indices are in the range [first, last]. If the second argument is omitted, the single line with index first is deleted.
get (first, last=None)	Returns a tuple containing the text of the lines with indices from first to last, inclusive. If the second argument is omitted, returns the text of the line closest to first.
index (i)	If possible, positions the visible part of the listbox so that the line containing index i is at the top of the widget.
insert (index, *elements)	Insert one or more new lines into the listbox before the line specified by index. Use END as the first argument if you want to add new lines to the end of the listbox.
nearest (y)	Return the index of the visible line closest to the y-coordinate y relative to the listbox widget.
see (index)	Adjust the position of the listbox so that the line referred to by index is visible.
size()	Returns the number of lines in the listbox.

xview()	To make the listbox horizontally scrollable, set the command option of the associated horizontal scrollbar to this method.
xview_moveto (fraction)	Scroll the listbox so that the leftmost fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
xview_scroll (number, what)	Scrolls the listbox horizontally. For the what argument, use either UNITS to scroll by characters, or PAGES to scroll by pages, that is, by the width of the listbox. The number argument tells how many to scroll.
yview()	To make the listbox vertically scrollable, set the command option of the associated vertical scrollbar to this method.
yview_moveto (fraction)	Scroll the listbox so that the top fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
yview_scroll (number, what)	Scrolls the listbox vertically. For the what argument, use either UNITS to scroll by lines, or PAGES to scroll by pages, that is, by the height of the listbox. The number argument tells how many to scroll.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *


import tkinter


top = Tk()


Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
```

```
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")

Lb1.pack()
top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Menubutton

A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget that can display the choices for that menubutton when the user clicks on it.

Syntax

Here is the simple syntax to create this widget-

```
w = Menubutton ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
activebackground	The background color when the mouse is over the menubutton.
activeforeground	The foreground color when the mouse is over the menubutton.

anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text.
bg	The normal background color displayed behind the label and indicator.
bitmap	To display a bitmap on the menubutton, set this option to a bitmap name.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	The cursor that appears when the mouse is over this menubutton.
direction	Set direction=LEFT to display the menu to the left of the button; use direction=RIGHT to display the menu to the right of the button; or use direction='above' to place the menu above the button.
disabledforeground	The foreground color shown on this menubutton when it is disabled.
fg	The foreground color when the mouse is not over the menubutton.
height	The height of the menubutton in lines of text (not pixels!). The default is to fit the menubutton's size to its contents.
highlightcolor	Color shown in the focus highlight when the widget has the focus.
image	To display an image on this menubutton,
justify	This option controls where the text is located when the text doesn't fill the menubutton: use justify=LEFT to left-justify the text (this is the default); use justify=CENTER to center it, or justify=RIGHT to right-justify.
menu	To associate the menubutton with a set of choices, set this option to the Menu object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument.

padx	How much space to leave to the left and right of the text of the menubutton. Default is 1.
pady	How much space to leave above and below the text of the menubutton. Default is 1.
relief	Selects three-dimensional border shading effects. The default is RAISED.
state	Normally, menubuttons respond to the mouse. Set state=DISABLED to gray out the menubutton and make it unresponsive.
text	To display text on the menubutton, set this option to the string containing the desired text. Newlines ("\n") within the string will cause line breaks.
textvariable	You can associate a control variable of class StringVar with this menubutton. Setting that control variable will change the displayed text.
underline	Normally, no underline appears under the text on the menubutton. To underline one of the characters, set this option to the index of that character.
width	The width of the widget in characters. The default is 20.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

import tkinter

top = Tk()

mb=  Menubutton ( top, text="condiments", relief=RAISED )
```

```
mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0 )
mb["menu"]  =  mb.menu

mayoVar  = IntVar()
ketchVar = IntVar()

mb.menu.add_checkbutton ( label="mayo",
                          variable=mayoVar )
mb.menu.add_checkbutton ( label="ketchup",
                          variable=ketchVar )
mb.pack()
top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Menu

The goal of this widget is to allow us to create all kinds of menus that can be used by our applications. The core functionality provides ways to create three menu types: pop-up, toplevel and pull-down.

It is also possible to use other extended widgets to implement new types of menus, such as the *OptionMenu* widget, which implements a special type that generates a pop-up list of items within a selection.

Syntax

Here is the simple syntax to create this widget-

```
w = Menu ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
activebackground	The background color that will appear on a choice when it is under the mouse.
activeborderwidth	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel.
activeforeground	The foreground color that will appear on a choice when it is under the mouse.
bg	The background color for choices not under the mouse.
bd	The width of the border around all the choices. Default is 1.
cursor	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off.
disabledforeground	The color of the text for items whose state is DISABLED.
font	The default font for textual choices.
fg	The foreground color used for choices not under the mouse.
postcommand	You can set this option to a procedure, and that procedure will be called every time someone brings up this menu.
relief	The default 3-D effect for menus is relief=RAISED.
image	To display an image on this menubutton.
selectcolor	Specifies the color displayed in checkbuttons and radiobuttons when they are selected.
tearoff	Normally, a menu can be torn off, the first position (position 0) in the list of choices is occupied by the tear-off element, and the additional choices are added starting at position 1. If you set

	tearoff=0, the menu will not have a tear-off feature, and choices will be added starting at position 0.
title	Normally, the title of a tear-off menu window will be the same as the text of the menubutton or cascade that lead to this menu. If you want to change the title of that window, set the title option to that string.

Methods

These methods are available on Menu objects-

Option	Description
add_command (options)	Adds a menu item to the menu.
add_radiobutton(options)	Creates a radio button menu item.
add_checkbutton(options)	Creates a check button menu item.
add_cascade(options)	Creates a new hierarchical menu by associating a given menu to a parent menu
add_separator()	Adds a separator line to the menu.
add(type, options)	Adds a specific type of menu item to the menu.
delete(startindex [, endindex])	Deletes the menu items ranging from startindex to endindex.
entryconfig(index, options)	Allows you to modify a menu item, which is identified by the index, and change its options.
index(item)	Returns the index number of the given menu item label.
insert_separator (index)	Insert a new separator at the position specified by index.

invoke (index)	Calls the command callback associated with the choice at position index. If a checkbutton, its state is toggled between set and cleared; if a radiobutton, that choice is set.
type (index)	Returns the type of the choice specified by index: either "cascade", "checkbutton", "command", "radiobutton", "separator", or "tearoff".

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *
def donothing():
   filewin = Toplevel(root)
   button = Button(filewin, text="Do nothing button")
   button.pack()

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)

filemenu.add_separator()

filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)
```

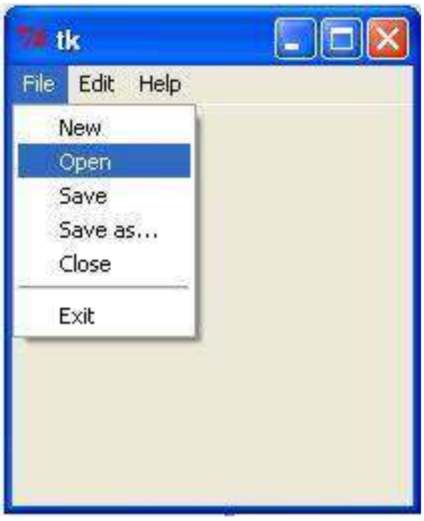
```
editmenu.add_separator()

editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
editmenu.add_command(label="Select All", command=donothing)

menubar.add_cascade(label="Edit", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)

root.config(menu=menubar)
root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Message

This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents.

Its functionality is very similar to the one provided by the Label widget, except that it can also automatically wrap the text, maintaining a given width or aspect ratio.

Syntax

Here is the simple syntax to create this widget-

```
w = Message ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
anchor	This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
bg	The normal background color displayed behind the label and indicator.
bitmap	Set this option equal to a bitmap or image object and the label will display that graphic.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
fg	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
height	The vertical dimension of the new frame.
image	To display a static image in the label widget, set this option to an image object.
justify	Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.

padx	Extra space added to the left and right of the text within the widget. Default is 1.
pady	Extra space added above and below the text within the widget. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
text	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
textvariable	To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

root = Tk()

var = StringVar()
label = Message( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
```

```
root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Radiobutton

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.

In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Syntax

Here is the simple syntax to create this widget-

```
w = Radiobutton ( master, option, ...  )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
activebackground	The background color when the mouse is over the radiobutton.
activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is anchor=CENTER.
bg	The normal background color behind the indicator and label.

bitmap	To display a monochrome image on a radiobutton, set this option to a bitmap.
borderwidth	The size of the border around the indicator part itself. Default is 2 pixels.
command	A procedure to be called every time the user changes the state of this radiobutton.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the radiobutton.
font	The font used for the text.
fg	The color used to render the text.
height	The number of lines (not pixels) of text on the radiobutton. Default is 1.
highlightbackground	The color of the focus highlight when the radiobutton does not have focus.
highlightcolor	The color of the focus highlight when the radiobutton has the focus.
image	To display a graphic image instead of text for this radiobutton, set this option to an image object.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER (the default), LEFT, or RIGHT.
padx	How much space to leave to the left and right of the radiobutton and text. Default is 1.
pady	How much space to leave above and below the radiobutton and text. Default is 1.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
selectcolor	The color of the radiobutton when it is set. Default is red.

selectimage	If you are using the image option to display a graphic instead of text when the radiobutton is cleared, you can set the selectimage option to a different image that will be displayed when the radiobutton is set.
state	The default is state=NORMAL, but you can set state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is ACTIVE.
text	The label displayed next to the radiobutton. Use newlines ("\n") to display multiple lines of text.
textvariable	To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
underline	You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
value	When a radiobutton is turned on by the user, its control variable is set to its current value option. If the control variable is an *IntVar*, give each radiobutton in the group a different integer value option. If the control variable is a*StringVar*, give each radiobutton a different string value option.
variable	The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an IntVar or a StringVar.
width	Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
wraplength	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Methods

These methods are available.

Methods	Description
deselect()	Clears (turns off) the radiobutton.

flash()	Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.
invoke()	You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.
select()	Sets (turns on) the radiobutton.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var, value=1,
                  command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Option 2", variable=var, value=2,
                  command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="Option 3", variable=var, value=3,
                  command=sel)
R3.pack( anchor = W)

label = Label(root)
label.pack()
root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Scale

The Scale widget provides a graphical slider object that allows you to select values from a specific scale.

Syntax

Here is the simple syntax to create this widget-

```
w = Scale ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
activebackground	The background color when the mouse is over the scale.
bg	The background color of the parts of the widget that are outside the trough.
bd	Width of the 3-d border around the trough and slider. Default is 2 pixels.
command	A procedure to be called every time the slider is moved. This procedure will be passed one argument, the new scale value. If the slider is moved rapidly, you may not get a callback for every possible position, but you'll certainly get a callback when it settles.

cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the scale.
digits	The way your program reads the current value shown in a scale widget is through a control variable. The control variable for a scale can be an IntVar, a DoubleVar (float), or a StringVar. If it is a string variable, the digits option controls how many digits to use when the numeric scale value is converted to a string.
font	The font used for the label and annotations.
fg	The color of the text used for the label and annotations.
from_	A float or integer value that defines one end of the scale's range.
highlightbackground	The color of the focus highlight when the scale does not have focus.
highlightcolor	The color of the focus highlight when the scale has the focus.
label	You can display a label within the scale widget by setting this option to the label's text. The label appears in the top left corner if the scale is horizontal, or the top right corner if vertical. The default is no label.
length	The length of the scale widget. This is the x dimension if the scale is horizontal, or the y dimension if vertical. The default is 100 pixels.
orient	Set orient=HORIZONTAL if you want the scale to run along the x dimension, or orient=VERTICAL to run parallel to the y-axis. Default is horizontal.
relief	Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
resolution	Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest

	increment of the scale's value. For example, if from_=-1.0 and to=1.0, and you set resolution=0.5, the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0.
showvalue	Normally, the current value of the scale is displayed in text form by the slider (above it for horizontal scales, to the left for vertical scales). Set this option to 0 to suppress that label.
sliderlength	Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the sliderlength option to your desired length.
state	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set state=DISABLED to make the widget unresponsive.
takefocus	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior.
tickinterval	To display periodic scale values, set this option to a number, and ticks will be displayed on multiples of that value. For example, if from_=0.0, to=1.0, and tickinterval=0.25, labels will be displayed along the scale at values 0.0, 0.25, 0.50, 0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.
to	A float or integer value that defines one end of the scale's range; the other end is defined by the from_ option, discussed above. The to value can be either greater than or less than the from_ value. For vertical scales, the to value defines the bottom of the scale; for horizontal scales, the right end.
troughcolor	The color of the trough.
variable	The control variable for this scale, if any. Control variables may be from class IntVar, DoubleVar (float), or StringVar. In the latter case, the numerical value will be converted to a string.
width	The width of the trough part of the widget. This is the x dimension for vertical scales and the y dimension if the scale has orient=HORIZONTAL. Default is 15 pixels.

Methods

Scale objects have these methods-

Methods	Description
get()	This method returns the current value of the scale.
set (value)	Sets the scale's value.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

def sel():
   selection = "Value = " + str(var.get())
   label.config(text = selection)

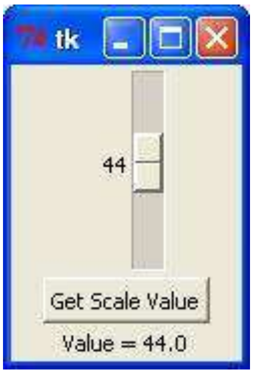
root = Tk()
var = DoubleVar()
scale = Scale( root, variable = var )
scale.pack(anchor=CENTER)

button = Button(root, text="Get Scale Value", command=sel)
button.pack(anchor=CENTER)

label = Label(root)
label.pack()

root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Scrollbar

This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text and Canvas. Note that you can also create horizontal scrollbars on Entry widgets.

Syntax

Here is the simple syntax to create this widget-

```
w = Scrollbar ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.

cursor	The cursor that appears when the mouse is over the scrollbar.
elementborderwidth	The width of the borders around the arrowheads and slider. The default is elementborderwidth=-1, which means to use the value of the borderwidth option.
highlightbackground	The color of the focus highlight when the scrollbar does not have focus.
highlightcolor	The color of the focus highlight when the scrollbar has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
jump	This option controls what happens when a user drags the slider. Normally (jump=0), every small drag of the slider causes the command callback to be called. If you set this option to 1, the callback isn't called until the user releases the mouse button.
orient	Set orient=HORIZONTAL for a horizontal scrollbar, orient=VERTICAL for a vertical one.
repeatdelay	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
repeatinterval	repeatinterval
takefocus	Normally, you can tab the focus through a scrollbar widget. Set takefocus=0 if you don't want this behavior.
troughcolor	The color of the trough.
width	Width of the scrollbar (its y dimension if horizontal, and its x dimension if vertical). Default is 16.

Methods

Scrollbar objects have these methods-

Methods	Description

get()	Returns two numbers (a, b) describing the current position of the slider. The a value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the b value gives the position of the right or bottom edge.
set (first, last)	To connect a scrollbar to another widget w, set w's xscrollcommand or yscrollcommand to the scrollbar's set() method. The arguments have the same meaning as the values returned by the get() method.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

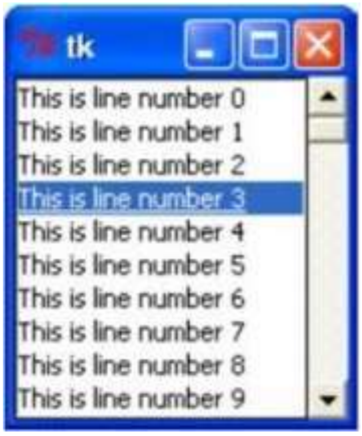
root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill=Y )

mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
   mylist.insert(END, "This is line number " + str(line))

mylist.pack( side = LEFT, fill = BOTH )
scrollbar.config( command = mylist.yview )

mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Text

Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font.

You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

Syntax

Here is the simple syntax to create this widget-

```
w = Text ( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
bg	The default background color of the text widget.
bd	The width of the border around the text widget. Default is 2 pixels.
cursor	The cursor that will appear when the mouse is over the text widget.
exportselection	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	The height of the widget in lines (not pixels!), measured according to the current font size.

highlightbackground	The color of the focus highlight when the text widget does not have focus.
highlightcolor	The color of the focus highlight when the text widget has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set highlightthickness=0 to suppress display of the focus highlight.
insertbackground	The color of the insertion cursor. Default is black.
insertborderwidth	Size of the 3-D border around the insertion cursor. Default is 0.
insertofftime	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
insertontime	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
insertwidth	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
padx	The size of the internal padding added to the left and right of the text area. Default is one pixel.
pady	The size of the internal padding added above and below the text area. Default is one pixel.
relief	The 3-D appearance of the text widget. Default is relief=SUNKEN.
selectbackground	The background color to use displaying selected text.
selectborderwidth	The width of the border to use around selected text.
spacing1	This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is 0.

spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
spacing3	This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is 0.
state	Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED, the text widget will not respond, and you won't be able to modify its contents programmatically either.
tabs	This option controls how tab characters position text.
width	The width of the widget in characters (not pixels!), measured according to the current font size.
wrap	This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too long will be broken at any character.
xscrollcommand	To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.
yscrollcommand	To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.

Methods

Text objects have these methods-

Methods & Description
delete(startindex [,endindex]) This method deletes a specific character or a range of text.
get(startindex [,endindex]) This method returns a specific character or a range of text.
index(index) Returns the absolute value of an index based on the given index.

insert(index [,string]...) This method inserts strings at the specified index location.
see(index) This method returns true if the text located at the index position is visible.

Text widgets support three distinct helper structures: Marks, Tabs, and Indexes-

Marks are used to bookmark positions between two characters within a given text. We have the following methods available when handling marks:

Methods & Description
index(mark) Returns the line and column location of a specific mark.
mark_gravity(mark [,gravity]) Returns the gravity of the given mark. If the second argument is provided, the gravity is set for the given mark.
mark_names() Returns all marks from the Text widget.
mark_set(mark, index) Informs a new position to the given mark.
mark_unset(mark) Removes the given mark from the Text widget.

Tags are used to associate names to regions of text which makes easy the task of modifying the display settings of specific text areas. Tags are also used to bind event callbacks to specific ranges of text.

Following are the available methods for handling tabs-

Methods & Description
tag_add(tagname, startindex[,endindex] ...) This method tags either the position defined by startindex, or a range delimited by the positions startindex and endindex.
tag_config You can use this method to configure the tag properties, which include, justify(center, left, or right), tabs(this property has the same functionality of the Text widget tabs's property), and underline(used to underline the tagged text).

tag_delete(tagname) This method is used to delete and remove a given tag.
tag_remove(tagname [,startindex[.endindex]] ...) After applying this method, the given tag is removed from the provided area without deleting the actual tag definition.

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

root = Tk()
text = Text(root)
text.insert(INSERT, "Hello.....")
text.insert(END, "Bye Bye.....")
text.pack()

text.tag_add("here", "1.0", "1.4")
text.tag_add("start", "1.8", "1.13")
text.tag_config("here", background="yellow", foreground="blue")
text.tag_config("start", background="black", foreground="green")
root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Toplevel

Toplevel widgets work as windows that are directly managed by the window manager. They do not necessarily have a parent widget on top of them.

Your application can use any number of top-level windows.

Syntax

Here is the simple syntax to create this widget-

```
w = Toplevel ( option, ... )
```

Parameters

options: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
bg	The background color of the window.
bd	Border width in pixels; default is 0.
cursor	The cursor that appears when the mouse is in this window.
class_	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
font	The default font for text inserted into the widget.
fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
height	Window height.
relief	Normally, a top-level window will have no 3-d borders around it. To get a shaded border, set the bd option larger that its default value of zero, and set the relief option to one of the constants.
width	The desired width of the window.

Methods

Toplevel objects have these methods-

Methods and Description
deiconify()

Displays the window, after using either the iconify or the withdraw methods.
frame() Returns a system-specific window identifier.
group(window) Adds the window to the window group administered by the given window.
iconify() Turns the window into an icon, without destroying it.
protocol(name, function) Registers a function as a callback which will be called for the given protocol.
iconify() Turns the window into an icon, without destroying it.
state() Returns the current state of the window. Possible values are normal, iconic, withdrawn and icon.
transient([master]) Turns the window into a temporary(transient) window for the given master or to the window's parent, when no argument is given.
withdraw() Removes the window from the screen, without destroying it.
maxsize(width, height) Defines the maximum size for this window.
minsize(width, height) Defines the minimum size for this window.
positionfrom(who) Defines the position controller.

resizable(width, height) Defines the resize flags, which control whether the window can be resized.
sizefrom(who) Defines the size controller.
title(string) Defines the window title.

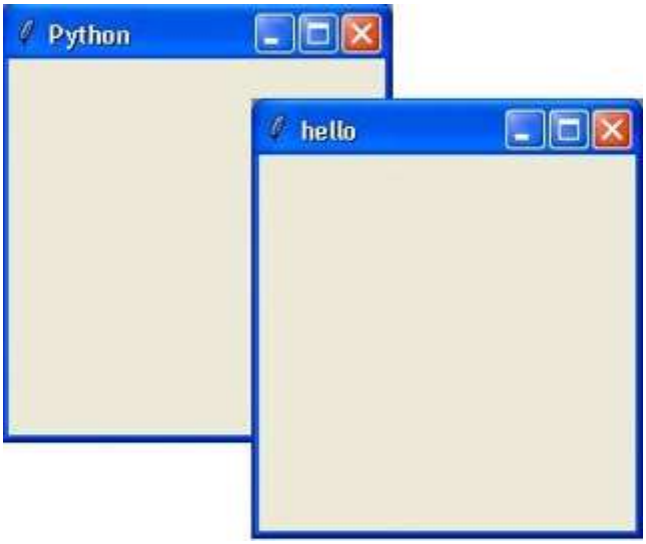
Example

Try following example yourself-

```
# !/usr/bin/python3
from tkinter import *

root = Tk()
root.title("hello")
top = Toplevel()
top.title("Python")
top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Spinbox

The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.

Syntax

Here is the simple syntax to create this widget-

```
w = Spinbox( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Options	Description
activebackground	The color of the slider and arrowheads when the mouse is over them.
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
command	A procedure to be called whenever the scrollbar is moved.
cursor	The cursor that appears when the mouse is over the scrollbar.
disabledbackground	The background color to use when the widget is disabled.
disabledforeground	The text color to use when the widget is disabled.
fg	Text color.
font	The font to use in this widget.
format	Format string. No default value.
from_	The minimum value. Used together with to to limit the spinbox range.

justify	Default is LEFT
relief	Default is SUNKEN.
repeatdelay	Together with repeatinterval, this option controls button auto-repeat. Both values are given in milliseconds.
repeatinterval	See repeatdelay.
state	One of NORMAL, DISABLED, or "readonly". Default is NORMAL.
textvariable	No default value.
to	See from.
validate	Validation mode. Default is NONE.
validatecommand	Validation callback. No default value.
values	A tuple containing valid values for this widget. Overrides from/to/increment.
vcmd	Same as validatecommand.
width	Widget width, in character units. Default is 20.
wrap	If true, the up and down buttons will wrap around.
xscrollcommand	Used to connect a spinbox field to a horizontal scrollbar. This option should be set to the set method of the corresponding scrollbar.

Methods

Spinbox objects have these methods-

Methods and Description
delete(startindex [,endindex]) This method deletes a specific character or a range of text.

get(startindex [,endindex]) This method returns a specific character or a range of text.
identify(x, y) Identifies the widget element at the given location.
index(index) Returns the absolute value of an index based on the given index.
insert(index [,string]...) This method inserts strings at the specified index location.
invoke(element) Invokes a spinbox button.

Example

Try the following example yourself-

```
from Tkinter import *

master = Tk()

w = Spinbox(master, from_=0, to=10)
w.pack()

mainloop()
```

When the above code is execduted, it produces the following result-



Tkinter PanedWindow

A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.

Each pane contains one widget and each pair of panes is separated by a moveable (via mouse movements) sash. Moving a sash causes the widgets on either side of the sash to be resized.

Syntax

Here is the simple syntax to create this widget-

```
w = PanedWindow( master, option, ... )
```

Parameters

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The color of the slider and arrowheads when the mouse is not over them.
bd	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
borderwidth	Default is 2.
cursor	The cursor that appears when the mouse is over the window.
handlepad	Default is 8.
handlesize	Default is 8.
height	No default value.
orient	Default is HORIZONTAL.

relief	Default is FLAT.
sashcursor	No default value.
sashrelief	Default is RAISED.
sashwidth	Default is 2.
showhandle	No default value
width	No default value.

Methods

PanedWindow objects have these methods-

Methods & Description
add(child, options) Adds a child window to the paned window.
get(startindex [,endindex]) This method returns a specific character or a range of text.
config(options) Modifies one or more widget options. If no options are given, the method returns a dictionary containing all current option values.

Example

Try the following example yourself. Here is how to create a 3-pane widget-

```
# !/usr/bin/python3
from tkinter import *
```

```
m1 = PanedWindow()
m1.pack(fill=BOTH, expand=1)

left = Entry(m1, bd=5)
m1.add(left)

m2 = PanedWindow(m1, orient=VERTICAL)
m1.add(m2)

top = Scale( m2, orient=HORIZONTAL)
m2.add(top)

bottom = Button(m2, text="OK")
m2.add(bottom)

mainloop()
```

When the above code is executed, it produces the following result-



Tkinter LabelFrame

A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.

This widget has the features of a frame plus the ability to display a label.

Syntax

Here is the simple syntax to create this widget-

```
w = LabelFrame( master, option, ... )
```

Parameters

- **master:** This represents the parent window.

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Option	Description
bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
font	The vertical dimension of the new frame.
height	The vertical dimension of the new frame.
labelAnchor	Specifies where to place the label.
highlightbackground	Color of the focus highlight when the frame does not have focus.
highlightcolor	Color shown in the focus highlight when the frame has the focus.
highlightthickness	Thickness of the focus highlight.
relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
text	Specifies a string to be displayed inside the widget.
width	Specifies the desired width for the window.

Example

Try the following example yourself. Here is how to create a labelframe widget-

```
# !/usr/bin/python3
from tkinter import *
```

```
root = Tk()

labelframe = LabelFrame(root, text="This is a LabelFrame")
labelframe.pack(fill="both", expand="yes")

left = Label(labelframe, text="Inside the LabelFrame")
left.pack()

root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter tkMessageBox

The tkMessageBox module is used to display message boxes in your applications. This module provides a number of functions that you can use to display an appropriate message.

Some of these functions are showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, and askretryignore.

Syntax

Here is the simple syntax to create this widget-

```
tkMessageBox.FunctionName(title, message [, options])
```

Parameters

- **FunctionName:** This is the name of the appropriate message box function.
- **title:** This is the text to be displayed in the title bar of a message box.
- **message:** This is the text to be displayed as a message.
- **options:** options are alternative choices that you may use to tailor a standard message box. Some of the options that you can use are default and parent. The

default option is used to specify the default button, such as ABORT, RETRY, or IGNORE in the message box. The parent option is used to specify the window on top of which the message box is to be displayed.

You could use one of the following functions with dialogue box-

- showinfo()
- showwarning()
- showerror ()
- askquestion()
- askokcancel()
- askyesno ()
- askretrycancel ()

Example

Try the following example yourself-

```
# !/usr/bin/python3
from tkinter import *

from tkinter import messagebox

top = Tk()
top.geometry("100x100")
def hello():
   messagebox.showinfo("Say Hello", "Hello World")

B1 = Button(top, text = "Say Hello", command = hello)
B1.place(x=35,y=50)

top.mainloop()
```

When the above code is executed, it produces the following result-



Standard Attributes

Let us look at how some of the common attributes, such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors
- Relief styles
- Bitmaps
- Cursors

Let us study them briefly-

Tkinter Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units.

- If you set a dimension to an integer, it is assumed to be in pixels.
- You can specify units by setting a dimension to a string containing a number followed by.

Character	Description

c	Centimeters
i	Inches
m	Millimeters
p	Printer's points (about 1/72")

Length options

Tkinter expresses a length as an integer number of pixels. Here is the list of common length options-

- **borderwidth:** Width of the border which gives a three-dimensional look to the widget.
- **highlightthickness:** Width of the highlight rectangle when the widget has focus .
- **padX padY:** Extra space the widget requests from its layout manager beyond the minimum the widget needs to display its contents in the x and y directions.
- **selectborderwidth:** Width of the three-dimentional border around selected items of the widget.
- **wraplength:** Maximum line length for widgets that perform word wrapping.
- **height:** Desired height of the widget; must be greater than or equal to 1.
- **underline:** Index of the character to underline in the widget's text (0 is the first character, 1 the second one, and so on).
- **width:** Desired width of the widget.

Tkinter Colors

Tkinter represents colors with strings. There are two general ways to specify colors in Tkinter-

- You can use a string specifying the proportion of red, green and blue in hexadecimal digits. For example, "#fff" is white, "#000000" is black, "#000fff000" is pure green, and "#00ffff" is pure cyan (green plus blue).
- You can also use any locally defined standard color name. The colors "white", "black", "red", "green", "blue", "cyan", "yellow", and "magenta" will always be available.

Color options

The common color options are-

- **activebackground:** Background color for the widget when the widget is active.
- **activeforeground:** Foreground color for the widget when the widget is active.
- **background:** Background color for the widget. This can also be represented as *bg*.
- **disabledforeground:** Foreground color for the widget when the widget is disabled.
- **foreground:** Foreground color for the widget. This can also be represented as *fg*.
- **highlightbackground:** Background color of the highlight region when the widget has focus.
- **highlightcolor:** Foreground color of the highlight region when the widget has focus.
- **selectbackground:** Background color for the selected items of the widget.
- **selectforeground:** Foreground color for the selected items of the widget.

Tkinter Fonts

There may be up to three ways to specify type style.

Simple Tuple Fonts

As a tuple whose first element is the font family, followed by a size in points, optionally followed by a string containing one or more of the style modifiers bold, italic, underline and overstrike.

Example

- ("Helvetica", "16") for a 16-point Helvetica regular.
- ("Times", "24", "bold italic") for a 24-point Times bold italic.

Font object Fonts

You can create a "font object" by importing the tkFont module and using its Font class constructor –

```
import tkFont

font = tkFont.Font ( option, ... )
```

Here is the list of options-

- **family:** The font family name as a string.
- **size:** The font height as an integer in points. To get a font n pixels high, use -n.
- **weight:** "bold" for boldface, "normal" for regular weight.
- **slant:** "italic" for italic, "roman" for unslanted.
- **underline:** 1 for underlined text, 0 for normal.
- **overstrike:** 1 for overstruck text, 0 for normal.

Example

```
helv36 = tkFont.Font(family="Helvetica",size=36,weight="bold")
```

X Window Fonts

If you are running under the X Window System, you can use any of the X font names.

For example, the font named "-*-lucidatypewriter-medium-r-*-*-*-140-*-*-*-*-*-*" is the author's favorite fixed-width font for onscreen use. Use the*xfontsel* program to help you select pleasing fonts.

Tkinter Anchors

Anchors are used to define where text is positioned relative to a reference point.

Here is list of possible constants, which can be used for Anchor attribute.

- NW
- N
- NE
- W
- CENTER
- E
- SW
- S
- SE

For example, if you use CENTER as a text anchor, the text will be centered horizontally and vertically around the reference point.

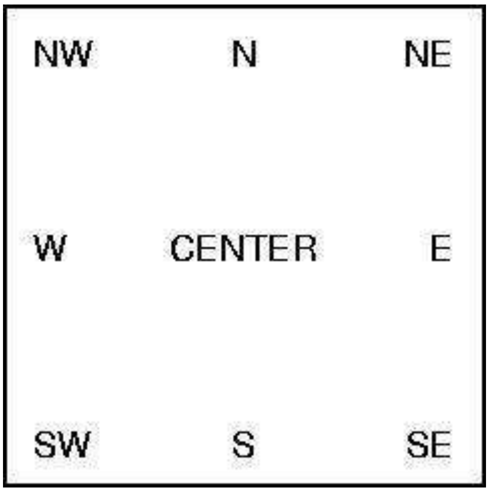
Anchor NW will position the text so that the reference point coincides with the northwest (top left) corner of the box containing the text.

Anchor W will center the text vertically around the reference point, with the left edge of the text box passing through that point, and so on.

If you create a small widget inside a large frame and use the anchor=SE option, the widget will be placed in the bottom right corner of the frame. If you used anchor=N instead, the widget would be centered along the top edge.

Example

The anchor constants are shown in this diagram-



Tkinter Relief styles

The relief style of a widget refers to certain simulated 3-D effects around the outside of the widget. Here is a screenshot of a row of buttons exhibiting all the possible relief styles-

Here is list of possible constants which can be used for relief attribute-

- FLAT
- RAISED
- SUNKEN
- GROOVE
- RIDGE

Example

```
# !/usr/bin/python3
from tkinter import *
import tkinter

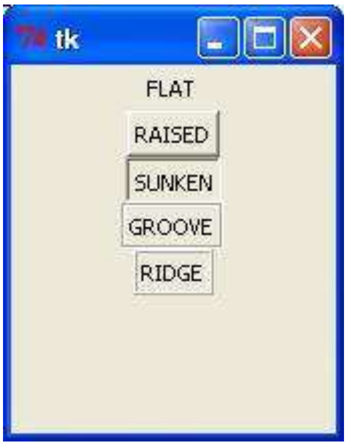
top = Tk()

B1 = Button(top, text ="FLAT", relief=FLAT )
```

```
B2 = Button(top, text ="RAISED", relief=RAISED )
B3 = Button(top, text ="SUNKEN", relief=SUNKEN )
B4 = Button(top, text ="GROOVE", relief=GROOVE )
B5 = Button(top, text ="RIDGE", relief=RIDGE )

B1.pack()
B2.pack()
B3.pack()
B4.pack()
B5.pack()
top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Bitmaps

This attribute to displays a bitmap. There are following type of bitmaps available-

- "error"
- "gray75"
- "gray50"
- "gray25"
- "gray12"
- "hourglass"
- "info"

- "questhead"
- "question"
- "warning"

Example

```
# !/usr/bin/python3
from tkinter import *
import tkinter

top = Tk()

B1 = Button(top, text ="error", relief=RAISED,\
                         bitmap="error")
B2 = Button(top, text ="hourglass", relief=RAISED,\
                         bitmap="hourglass")
B3 = Button(top, text ="info", relief=RAISED,\
                         bitmap="info")
B4 = Button(top, text ="question", relief=RAISED,\
                         bitmap="question")
B5 = Button(top, text ="warning", relief=RAISED,\
                         bitmap="warning")

B1.pack()
B2.pack()
B3.pack()
B4.pack()
B5.pack()
top.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter Cursors

Python Tkinter supports quite a number of different mouse cursors available. The exact graphic may vary according to your operating system.

Here is the list of interesting ones-

- "arrow"
- "circle"
- "clock"
- "cross"
- "dotbox"
- "exchange"
- "fleur"
- "heart"
- "heart"
- "man"
- "mouse"
- "pirate"
- "plus"
- "shuttle"
- "sizing"
- "spider"
- "spraycan"
- "star"
- "target"
- "tcross"
- "trek"
- "watch"

Example

Try the following example by moving cursor on different buttons-

```
# !/usr/bin/python3
from tkinter import *
import tkinter

top = Tk()

B1 = Button(top, text ="circle", relief=RAISED,\
                         cursor="circle")
B2 = Button(top, text ="plus", relief=RAISED,\
                         cursor="plus")
B1.pack()
B2.pack()
top.mainloop()
```

Geometry Management

All Tkinter widgets have access to the specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- The pack() Method - This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The grid() Method - This geometry manager organizes widgets in a table-like structure in the parent widget.
- The place() Method -This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Let us study the geometry management methods briefly –

Tkinter pack() Method

This geometry manager organizes widgets in blocks before placing them in the parent widget.

Syntax

```
widget.pack( pack_options )
```

Here is the list of possible options-

- **expand:** When set to true, widget expands to fill any space not otherwise used in widget's parent.

- **fill:** Determines whether widget fills any extra space allocated to it by the packer, or keeps its own minimal dimensions: NONE (default), X (fill only horizontally), Y (fill only vertically), or BOTH (fill both horizontally and vertically).

- **side:** Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.

Example

Try the following example by moving cursor on different buttons-

```
# !/usr/bin/python3
from tkinter import *

root = Tk()
frame = Frame(root)
frame.pack()

bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )
```

```
blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

When the above code is executed, it produces the following result-



Tkinter grid() Method

This geometry manager organizes widgets in a table-like structure in the parent widget.

Syntax

```
widget.grid( grid_options )
```

Here is the list of possible options-

- **column** : The column to put widget in; default 0 (leftmost column).
- **columnspan**: How many columns widgetoccupies; default 1.
- **ipadx, ipady** :How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- **padx, pady** : How many pixels to pad widget, horizontally and vertically, outside v's borders.
- **row**: The row to put widget in; default the first row that is still empty.
- **rowspan** : How many rowswidget occupies; default 1.
- **sticky** : What to do if the cell is larger than widget. By default, with sticky='', widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

Example

Try the following example by moving cursor on different buttons-

```
# !/usr/bin/python3
from  tkinter import *
root = Tk(  )
```

```
b=0
for r in range(6):
    for c in range(6):
        b=b+1
        Button(root, text=str(b),
            borderwidth=1 ).grid(row=r,column=c)
root.mainloop()
```

This would produce the following result displaying 12 labels arrayed in a 3 x 4 grid-



Tkinter place() Method

This geometry manager organizes widgets by placing them in a specific position in the parent widget.

Syntax

```
widget.place( place_options )
```

Here is the list of possible options-

- **anchor** : The exact spot of widget other options refer to: may be N, E, S, W, NE, NW, SE, or SW, compass directions indicating the corners and sides of widget; default is NW (the upper left corner of widget)

- **bordermode** : INSIDE (the default) to indicate that other options refer to the parent's inside (ignoring the parent's border); OUTSIDE otherwise.

- **height, width :** Height and width in pixels.

- **relheight, relwidth :** Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.

- **relx, rely :** Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.

- **x, y :** Horizontal and vertical offset in pixels.

Example

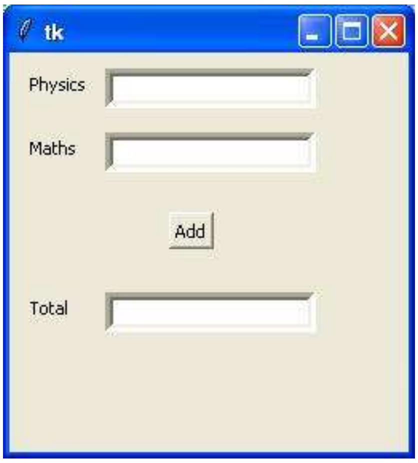
Try the following example by moving cursor on different buttons-

```
# !/usr/bin/python3
ffrom tkinter import *
top = Tk()
L1 = Label(top, text="Physics")
L1.place(x=10,y=10)
E1 = Entry(top, bd =5)
E1.place(x=60,y=10)
L2=Label(top,text="Maths")
L2.place(x=10,y=50)
E2=Entry(top,bd=5)
E2.place(x=60,y=50)

L3=Label(top,text="Total")
L3.place(x=10,y=150)
E3=Entry(top,bd=5)
E3.place(x=60,y=150)

B = Button(top, text ="Add")
B.place(x=100, y=100)
top.geometry("250x250+10+10")
top.mainloop()
```

When the above code is executed, it produces the following result-



28. Python 3 – Extension Programming with C

Any code that you write using any compiled language like C, C++, or Java can be integrated or imported into another Python script. This code is considered as an "extension."

A Python extension module is nothing more than a normal C library. On Unix machines, these libraries usually end in **.so** (for shared object). On Windows machines, you typically see **.dll** (for dynamically linked library).

Pre-Requisites for Writing Extensions

To start writing your extension, you are going to need the Python header files.

- On Unix machines, this usually requires installing a developer-specific package such as python2.5-dev.
- Windows users get these headers as part of the package when they use the binary Python installer.

Additionally, it is assumed that you have a good knowledge of C or C++ to write any Python Extension using C programming.

First look at a Python Extension

For your first look at a Python extension module, you need to group your code into four parts-

- The header file *Python.h*.
- The C functions you want to expose as the interface from your module.
- A table mapping the names of your functions as Python developers see them as C functions inside the extension module.
- An initialization function.

The Header File Python.h

You need to include Python.h header file in your C source file, which gives you the access to the internal Python API used to hook your module into the interpreter.

Make sure to include Python.h before any other headers you might need. You need to follow the includes with the functions you want to call from Python.

The C Functions

The signatures of the C implementation of your functions always takes one of the following three forms-

```
static PyObject *MyFunction( PyObject *self, PyObject *args );


static PyObject *MyFunctionWithKeywords(PyObject *self,
                                 PyObject *args,
                                 PyObject *kw);


static PyObject *MyFunctionWithNoArgs( PyObject *self );
```

Each one of the preceding declarations returns a Python object. There is no such thing as a *void* function in Python as there is in C. If you do not want your functions to return a value, return the C equivalent of Python's **None** value. The Python headers define a macro, Py_RETURN_NONE, that does this for us.

The names of your C functions can be whatever you like as they are never seen outside of the extension module. They are defined as *static* function.

Your C functions usually are named by combining the Python module and function names together, as shown here –

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Do your stuff here. */
   Py_RETURN_NONE;
}
```

This is a Python function called *func* inside the module *module*. You will be putting pointers to your C functions into the method table for the module that usually comes next in your source code.

The Method Mapping Table

This method table is a simple array of PyMethodDef structures. That structure looks something like this-

```
struct PyMethodDef {
   char *ml_name;
   PyCFunction ml_meth;
   int ml_flags;
   char *ml_doc;
};
```

Here is the description of the members of this structure-

- **ml_name:** This is the name of the function as the Python interpreter presents when it is used in Python programs.

- **ml_meth:** This is the address of a function that has any one of the signatures, described in the previous section.

- **ml_flags:** This tells the interpreter which of the three signatures ml_meth is using.

  - This flag usually has a value of METH_VARARGS.

  - This flag can be bitwise OR'ed with METH_KEYWORDS if you want to allow keyword arguments into your function.

  - This can also have a value of METH_NOARGS that indicates you do not want to accept any arguments.

- **ml_doc:** This is the docstring for the function, which could be NULL if you do not feel like writing one.

This table needs to be terminated with a sentinel that consists of NULL and 0 values for the appropriate members.

Example

For the above-defined function, we have the following method mapping table-

```
static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { NULL, NULL, 0, NULL }
};
```

The Initialization Function

The last part of your extension module is the initialization function. This function is called by the Python interpreter when the module is loaded. It is required that the function be named **init*Module***, where *Module* is the name of the module.

The initialization function needs to be exported from the library you will be building. The Python headers define PyMODINIT_FUNC to include the appropriate incantations for that to happen for the particular environment in which we are compiling. All you have to do is use it when defining the function.

Your C initialization function generally has the following overall structure-

```
PyMODINIT_FUNC initModule() {
   Py_InitModule3(func, module_methods, "docstring...");
}
```

Here is the description of Py_InitModule3 function-

- **func:** This is the function to be exported.
- **module_methods:** This is the mapping table name defined above.
- **docstring:** This is the comment you want to give in your extension.

Putting all this together, it looks like the following-

```
#include <Python.h>

static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Do your stuff here. */
   Py_RETURN_NONE;
}

static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC initModule() {
   Py_InitModule3(func, module_methods, "docstring...");
}
```

Example

A simple example that makes use of all the above concepts-

```
#include <Python.h>

static PyObject* helloworld(PyObject* self)
{
    return Py_BuildValue("s", "Hello, Python extensions!!");
}

static char helloworld_docs[] =
    "helloworld( ): Any message you want to put here!!\n";

static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld,
     METH_NOARGS, helloworld_docs},
```

```
    {NULL}
};

void inithelloworld(void)
{
    Py_InitModule3("helloworld", helloworld_funcs,
                   "Extension module example!");
}
```

Here the *Py_BuildValue* function is used to build a Python value. Save above code in hello.c file. We would see how to compile and install this module to be called from Python script.

Building and Installing Extensions

The *distutils* package makes it very easy to distribute Python modules, both pure Python and extension modules, in a standard way. Modules are distributed in the source form, built and installed via a setup script usually called *setup.py*as.

For the above module, you need to prepare the following setup.py script –

```
from distutils.core import setup, Extension
setup(name='helloworld', version='1.0',  \
      ext_modules=[Extension('helloworld', ['hello.c'])])
```

Now, use the following command, which would perform all needed compilation and linking steps, with the right compiler and linker commands and flags, and copies the resulting dynamic library into an appropriate directory-

```
$ python setup.py install
```

On Unix-based systems, you will most likely need to run this command as root in order to have permissions to write to the site-packages directory. This usually is not a problem on Windows.

Importing Extensions

Once you install your extensions, you would be able to import and call that extension in your Python script as follows-

```
#!/usr/bin/python3
import helloworld

print helloworld.helloworld()
```

This would produce the following result-

```
Hello, Python extensions!!
```

Passing Function Parameters

As you will most likely want to define functions that accept arguments, you can use one of the other signatures for your C functions. For example, the following function, that accepts some number of parameters, would be defined like this-

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Parse args and do something interesting here. */
   Py_RETURN_NONE;
}
```

The method table containing an entry for the new function would look like this-

```
static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { "func", module_func, METH_VARARGS, NULL },
   { NULL, NULL, 0, NULL }
};
```

You can use the API *PyArg_ParseTuple* function to extract the arguments from the one PyObject pointer passed into your C function.

The first argument to PyArg_ParseTuple is the args argument. This is the object you will be *parsing*. The second argument is a format string describing the arguments as you expect them to appear. Each argument is represented by one or more characters in the format string as follows.

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   int i;
   double d;
   char *s;

   if (!PyArg_ParseTuple(args, "ids", &i, &d, &s)) {
      return NULL;
   }

   /* Do something interesting here. */
   Py_RETURN_NONE;
}
```

Compiling the new version of your module and importing it enables you to invoke the new function with any number of arguments of any type-

```
module.func(1, s="three", d=2.0)
module.func(i=1, d=2.0, s="three")
```

```
module.func(s="three", d=2.0, i=1)
```

You can probably come up with even more variations.

The PyArg_ParseTuple Function

Here is the standard signature for the **PyArg_ParseTuple** function=

```
int PyArg_ParseTuple(PyObject* tuple,char* format,...)
```

This function returns 0 for errors, and a value not equal to 0 for success. Tuple is the PyObject* that was the C function's second argument. Here *format* is a C string that describes mandatory and optional arguments.

Here is a list of format codes for the **PyArg_ParseTuple** function-

Code	C type	Meaning
c	char	A Python string of length 1 becomes a C char.
d	double	A Python float becomes a C double.
f	float	A Python float becomes a C float.
i	int	A Python int becomes a C int.
l	long	A Python int becomes a C long.
L	long long	A Python int becomes a C long long
O	PyObject*	Gets non-NULL borrowed reference to Python argument.
s	char*	Python string without embedded nulls to C char*.
s#	char*+int	Any Python string to C address and length.
t#	char*+int	Read-only single-segment buffer to C address and length.
u	Py_UNICODE*	Python Unicode without embedded nulls to C.
u#	Py_UNICODE*+int	Any Python Unicode C address and length.

w#	char*+int	Read/write single-segment buffer to C address and length.
z	char*	Like s, also accepts None (sets C char* to NULL).
z#	char*+int	Like s#, also accepts None (sets C char* to NULL).
(...)	as per ...	A Python sequence is treated as one argument per item.
|		The following arguments are optional.
:		Format end, followed by function name for error messages.
;		Format end, followed by entire error message text.

Returning Values

Py_BuildValue takes in a format string much like *PyArg_ParseTuple* does. Instead of passing in the addresses of the values you are building, you pass in the actual values. Here is an example showing how to implement an add function-

```
static PyObject *foo_add(PyObject *self, PyObject *args) {
   int a;
   int b;

   if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
      return NULL;
   }
   return Py_BuildValue("i", a + b);
}
```

This is what it would look like if implemented in Python-

```
def add(a, b):
   return (a + b)
```

You can return two values from your function as follows. This would be captured using a list in Python.

```
static PyObject *foo_add_subtract(PyObject *self, PyObject *args) {
   int a;
   int b;
```

```
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    return Py_BuildValue("ii", a + b, a - b);
}
```

This is what it would look like if implemented in Python-

```
def add_subtract(a, b):
    return (a + b, a - b)
```

The Py_BuildValue Function

Here is the standard signature for **Py_BuildValue** function-

```
PyObject* Py_BuildValue(char* format,...)
```

Here *format* is a C string that describes the Python object to build. The following arguments of *Py_BuildValue* are C values from which the result is built. The*PyObject** result is a new reference.

The following table lists the commonly used code strings, of which zero or more are joined into a string format.

Code	C type	Meaning
c	char	A C char becomes a Python string of length 1.
d	double	A C double becomes a Python float.
f	float	A C float becomes a Python float.
i	int	A C int becomes a Python int.
l	long	A C long becomes a Python int.
N	PyObject*	Passes a Python object and steals a reference.
O	PyObject*	Passes a Python object and INCREFs it as normal.
O&	convert+void*	Arbitrary conversion

s	char*	C 0-terminated char* to Python string, or NULL to None.
s#	char*+int	C char* and length to Python string, or NULL to None.
u	Py_UNICODE*	C-wide, null-terminated string to Python Unicode, or NULL to None.
u#	Py_UNICODE*+int	C-wide string and length to Python Unicode, or NULL to None.
w#	char*+int	Read/write single-segment buffer to C address and length.
z	char*	Like s, also accepts None (sets C char* to NULL).
z#	char*+int	Like s#, also accepts None (sets C char* to NULL).
(...)	as per ...	Builds Python tuple from C values.
[...]	as per ...	Builds Python list from C values.
{...}	as per ...	Builds Python dictionary from C values, alternating keys and values.

Code {...} builds dictionaries from an even number of C values, alternately keys and values. For example, Py_BuildValue("{issi}",23,"zig","zag",42) returns a dictionary like Python's {23:'zig','zag':42}.